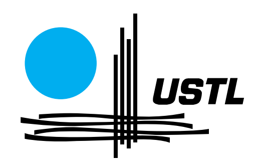






Calculs longs et partage des ressources processeur dans les systèmes multi-agents cognitifs

THÈSE

présentée et soutenue publiquement le 30 mars 2007

pour l'obtention du

Doctorat de l'Université des Sciences et Technologies de Lille
(spécialité informatique)

par

Cédric Dinont

Composition du jury

Rapporteurs :	Amal El Fallah Seghrouchni, Professeur	LIP6, Université Pierre et Marie Curie
	Alessandro Ricci, Docteur	DEIS, Université de Bologne
Examinateurs :	Laurence Duchien, Professeur	LIFL, Université Lille I
	Pierre Marquis, Professeur	CRIL, Université d'Artois
	Emmanuel Druon, Docteur	ISEN
	Patrick Taillibert	Thales Division Aéronautique
Directeur :	Philippe Mathieu, Professeur	LIFL, Université de Lille I

UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE
Laboratoire d'Informatique Fondamentale de Lille — UPRESA 8022
U.F.R. d'I.E.E.A. – Bât. M3 – 59655 VILLENEUVE D'ASCQ CEDEX
Tél. : +33 (0)3 28 77 85 41 – Télécopie : +33 (0)3 28 77 85 37 – email : direction@lifl.fr



Mis en page et composé avec LaTeX et la classe thlifl.

Remerciements

Je tiens ici à remercier les trois « chefs » qui m'ont suivi tout au long de cette thèse.

Philippe, tu as tout d'abord été un enseignant passionné et passionnant. Comme directeur de thèse, j'ai beaucoup apprécié ton enthousiasme permanent et ta persévérance pour améliorer la clarté de mon discours sur ces travaux (notamment ces fameuses trois questions, si essentielles).

Je remercie Patrick pour avoir accepté de me prendre en stage de DEA et pour m'avoir proposé le sujet de cette thèse. J'apprécie ta manière de conduire des recherches, notamment au travers de séances de brainstorming parfois déstabilisantes mais ô combien importantes pour faire apparaître de nouvelles idées, identifier les failles des propositions et apporter petit à petit les améliorations nécessaires.

Emmanuel, tu as été très disponible pour me conseiller efficacement tout au long de cette thèse. Tu as su me soutenir et m'encourager aux moments où j'en avais besoin. Je suis impatient de me mettre à travailler sur les nouveaux projets qui nous intéressent tous deux.

Merci aussi aux autres membres de l'équipe SMAC actuels ou passés : Jean-Paul, Jean-Christophe, Yann, Bruno, Sébastien, Damien, Julien, Marie-Hélène, Laëtitia, Tony, Maxime et tous les autres stagiaires ou étudiants qui ont fait un petit bout de chemin avec nous. Je n'oublierai pas l'ambiance qu'il peut y avoir dans ce bureau, surtout dès que Bruno ou Philippe sont là ! J'ai également de très bons souvenirs des repas d'équipe du jeudi midi.

Les membres du département informatique de l'ISEN m'ont également beaucoup apporté : Emmanuel, Frédéric, les trois Dominique et Pascal. Nous avons une vision commune de la pédagogie qui nous pousse à toujours améliorer notre pratique de l'enseignement. Je suis impressionné par notre capacité à nous entraider et à nous soutenir dans les moments où nous sommes débordés. Je suis extrêmement heureux que l'aventure puisse continuer après cette thèse.

Merci surtout à Cécile qui m'a soutenu durant ces trois années. Je n'aurais certainement pas pu aller au bout de ce travail sans ses encouragements.

Ce travail de thèse a été financé par Thales Systèmes Aéroportés, le conseil régional du Nord-Pas-de-Calais et l'Institut Supérieur de l'Électronique et du Numérique.

À mon fils Mathieu conçu durant la rédaction de cette thèse.

Résumé

Nous posons dans cette thèse le problème du respect de délais dans les systèmes multi-agents cognitifs. Les systèmes de gestion du temps utilisés dans ce cadre doivent répondre à un certain nombre de contraintes principalement liées aux spécificités attribuées aux agents. Ainsi, le système d'attribution des ressources processeur chargé de garantir des délais aux tâches de calcul doit respecter l'autonomie et la conscience du temps de nos agents.

Notre but est de proposer aux programmeurs de systèmes à base d'agents intelligents des outils permettant de gérer l'exécution simultanée d'algorithmes complexes, en particulier ceux issus des recherches en Intelligence Artificielle. Ces derniers posent souvent des problèmes d'intégration tant leur temps d'exécution peuvent être longs et variables. En effet, des agents qui mettraient en œuvre ce type d'algorithmes sans outils spécifiques pourraient perdre « conscience » et seraient incapables de rester en phase avec leur environnement.

Pour résoudre ces problèmes, nous proposons d'introduire dans les systèmes multi-agents une nouvelle classe d'entités qui jouent le rôle d'outils de calcul que les agents peuvent utiliser pour externaliser leurs calculs longs et ainsi rester à l'écoute de leur environnement. Nous étendons pour cela le concept d'artifact proposé par Omicini, Ricci et Viroli en 2004. Nous proposons également d'utiliser un artifact de coordination qui permet d'attribuer les ressources processeur en fonction des contraintes de chaque agent. Lorsque cet artifact ne peut respecter toutes les contraintes posées, les agents peuvent, par son intermédiaire, résoudre les conflits en sacrifiant une partie des ressources qui leur ont été attribuées. Les propositions ont été mises en œuvre sur la plateforme de développement d'agents ALBA mise en place au sein de Thales Division Aéronautique et évaluées en réimplémentant une application existante.

Mots-clés: Systèmes multi-agents, partage de processeur, ordonnancement, interaction, coordination, sacrifice, artifacts, algèbre de processus.

Abstract

In this thesis, we raise the problem of meeting deadlines in cognitive multi-agent systems. Time management systems used within this context must meet constraints mainly linked with the properties attributed to agents. Thus, the CPU resources allocation system that ensures deadlines meeting have to respect the autonomy and time-awareness of our agents.

Our aim is to propose tools allowing to manage the simultaneous execution of complex algorithms, especially algorithms stemming from the Artificial Intelligence field. Those algorithms are often hard to integrate in multi-agent systems as their execution times can be long and hard to predict. Indeed, agents which would use this kind of algorithms without specific tools could lose "consciousness" and would be unable to stay aware of their environment.

To solve these problems, we propose to introduce in multi-agent systems a new entity that can be viewed as a computational tool the agents can use to do their heavy computations and thus staying aware of their environment. For this purpose, we extend the artifact concept proposed by Omicini, Ricci and Viroli in 2004. We also propose to use a coordination artifact which allows to attribute CPU resources according to the constraints of every agent. When there is an inconsistency in the set of constraints, agents can use the coordination artifact as an intermediary to solve it. Our propositions have been implemented using the agent development platform (ALBA) designed at Thales Airborne Division and validated by re-engineering an existing application.

Keywords: Multi-agent systems, CPU sharing, scheduling, interaction, coordination, sacrifice, artifacts, process algebra.

Table des matières

Table des figures

Chapitre 1

Introduction

Cette thèse se propose d'aborder la gestion du temps dans les systèmes multi-agents en regardant ce problème à l'intersection de l'Intelligence Artificielle et du génie logiciel. Les algorithmes issus des recherches en Intelligence Artificielle posent souvent des problèmes d'intégration notamment car leurs temps d'exécution peuvent être longs et variables. Le problème crucial dans ces situations de concurrence est la gestion des délais que les agents ont à respecter.

Notre but est donc de proposer dans ce cadre des outils permettant d'intégrer facilement des algorithmes, de gérer leur exécution simultanée et de contrôler leur temps d'exécution. La thèse que nous soutenons ici est que pour arriver à cet objectif, il faut distinguer les agents, qui sont des entités douées de raisonnement, des artifacts de calcul, qui sont des entités passives que les agents peuvent utiliser comme outils pour atteindre leurs buts. Nous proposons une classification des modes d'emploi associés aux artifacts de calcul en fonction des propriétés temporelles des algorithmes qu'ils encapsulent. Notre travail apporte également un système qui gère l'attribution des ressources processeur de manière spécifique et qui est extensible à des systèmes multi-agents distribués.

Ce mémoire a été réalisé au sein du Laboratoire d'Informatique Fondamentale de Lille (LIFL) en collaboration avec la société Thales Division Aéronautique et l'Institut Supérieur de l'Électronique et du Numérique (ISEN).

1.1 Le but de l'IA : concevoir des agents

La question de l'intelligence des machines a été soulevée avant même que le premier ordinateur ne soit construit. La vision première, sous-tendue par le fameux test de Turing, était qu'une machine pourrait être considérée comme intelligente si son comportement vu par un observateur pouvait être confondu avec celui d'un Homme. Cette vision a reçu de nombreuses critiques, en particulier parce qu'une machine pourrait réussir à passer le test de Turing en trouvant les réponses aux questions qu'on lui pose dans une immense table, ce qui a peu d'intérêt du point de vue de la recherche.

Par la suite, c'est une vision plus scientifique qui a vu le jour. Le but de copier le comportement humain a été mis de côté pour être remplacé par l'étude systématique de deux thématiques : la représentation et la manipulation des connaissances ainsi que la recherche de solutions à un problème dans un espace contenant énormément d'éléments.

Le nom Intelligence Artificielle pour ce nouveau domaine de recherche a été choisi lors de la réunion d'un groupe de travail organisé par John McCarthy au Darthmouth College durant l'été 1956. Newell et Simon y ont présenté leur programme nommé Logic Theorist qui a été capable de trouver une preuve pour un théorème qui était plus courte que celle proposée par Russell et Whitehead dans *Principia Mathematica*. Malheureusement, l'article que Newell et Simon avaient fait co-signé par leur programme Logic Theorist n'a pas été accepté par les éditeurs du Journal of Symbolic Logic ! Cela montre bien l'ambiguïté liée au choix de l'expression Intelligence Artificielle. Beaucoup pensent que « rationalité computationelle » aurait été un meilleur choix.

Quoi qu'il en soit, la communauté IA s'est ensuite moins préoccupée du choix des termes, des implications philosophiques des résultats possibles et s'est structurée autour de la résolution de sous-problèmes :

- la planification,
- l'apprentissage automatique,
- le diagnostic et les systèmes experts,
- les jeux : échecs, jeux de cartes, jeux de mots, jeu de go,
- la reconnaissance de formes et le traitement automatique des langues,
- ...

Les résultats de l'IA ont été fluctuants : tantôt extraordinaires, avec les premiers systèmes d'aide à la résolution de problèmes mathématiques, de reconnaissance de la parole, de résolution de puzzles, . . . ; tantôt décevants, lorsque les chercheurs s'apercevaient qu'ils ne pouvaient résoudre des problèmes que dans des micro-mondes tant la combinatoire est importante dans des mondes réels.

Depuis le milieu des années 1990, certainement grâce à des résultats convaincants en planification notamment, un regain d'intérêt est apparu pour la constitution d'un « agent » intelligent utilisant conjointement les résultats des différents sous-domaines de l'IA. De ce point de vue, un agent serait un système d'IA complet capable d'utiliser ses différentes capacités intelligentes de base pour évoluer dans un environnement complexe. Ainsi, un agent peut être basiquement vu de l'extérieur comme une entité qui agit[1] dans un environnement au travers de capteurs et d'actionneurs (figure 1.1). Le programme d'un agent doit donc décider des actions à réaliser dans l'environnement en fonction des informations qu'il en perçoit. On parle aussi de fonction de l'agent.

Il est souvent remarqué qu'il n'y a pas encore de consensus sur la définition des propriétés que doit avoir une entité informatique pour pouvoir être nommée agent. Cette difficulté à aboutir à un consensus provient de la diversité des environnements dans lesquels nous pouvons plonger les agents. Les définitions à donner aux propriétés d'autonomie, de réactivité, de pro-activité, de prise en compte des changements dans l'environnement (extraversion) et donc la définition même de la notion d'agent varient avec l'environnement dans lequel se trouvent les agents.

[1] Agent vient du Latin *agere*, agir/faire.

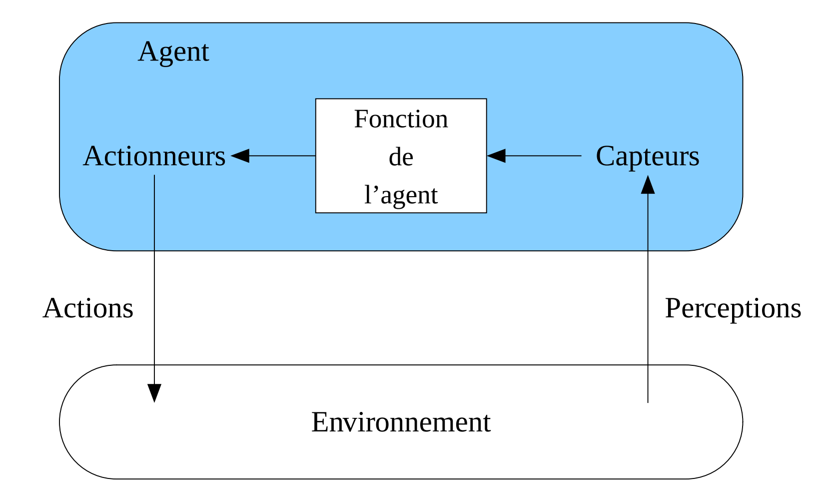


FIG. 1.1 – Un agent interagit avec son environnement au travers de capteurs et d'actionneurs.

Nous décrivons dans cette thèse une manière particulière de voir l'environnement des agents. Celui-ci est vu au travers des problèmes que posent le temps et le partage des ressources de calcul dans la conception de la fonction d'agents informatiques ayant des calculs lourds à réaliser. L'environnement est constitué des processus des agents s'exécutant dans le SMA. Le support de l'environnement étant l'ensemble des processeurs utilisés par le SMA, nous prendrons en compte dans sa représentation les ressources de calcul proposées par les processeurs. Nous aborderons quand cela sera nécessaire les questions de terminologie. Ce sera par exemple le cas lorsqu'il nous faudra parler de l'autonomie, de la pro-activité ou de l'extraversion des agents. Nous donnerons à ces termes des définitions adaptées à l'environnement particulier décrit ici.

Lorsque l'on parle de la gestion du temps pour des agents intelligents, il est inévitable d'aborder la comparaison avec la manière dont l'Homme gère le temps. Notre capacité la plus étonnante est certainement la façon avec laquelle nous sommes capables de gérer plusieurs contextes simultanément et de passer d'un contexte à l'autre automatiquement dès que l'on perçoit un signal d'attention pour un contexte particulier. Par exemple, nous sommes capables de nous arrêter de travailler pour répondre au téléphone, de changer de contexte pour s'adapter à la conversation que l'on tient au téléphone puis de revenir à notre contexte de travail lorsque la conversation téléphonique est terminée.

Nous prendrons garde de ne pas tomber dans les travers dont nous parlions précédemment et de n'utiliser la métaphore humaine que lorsqu'elle sera pertinente. Rappelons nous simplement que l'environnement dans lequel nous vivons n'est pas celui dans lequel se trouvent nos agents.

1.2 Le génie logiciel : l'amélioration des techniques de programmation

Le domaine du génie logiciel est apparu dans les années 1970 lorsque les premiers projets informatiques d'envergure ont affronté d'énormes difficultés pour arriver à leur terme. Le but est de

proposer des méthodes, techniques et outils pour concevoir et réaliser des systèmes informatiques de plus en plus complexes. Le génie logiciel prône des principes comme :

- **la simplicité.** Cette citation de Dave Small dans ST Magazine résume bien ce principe : « Un langage de programmation est censé être une façon conventionnelle de donner des ordres à un ordinateur. Il n'est pas censé être obscur, bizarre et plein de pièges subtils (ça ce sont les attributs de la magie) ».
- **la productivité.** Les traitements simples et récurrents ne doivent pas prendre beaucoup de temps à implémenter. Il faut également faciliter la mise au point de programme complexes grâce à un agencement de traitements de base.
- **l'efficacité.** Le programmeur doit disposer de structures de données, de bibliothèques et de langages permettant de créer des programmes qui utilisent au mieux les ressources disponibles à l'exécution.
- **la maintenabilité.** Les besoins qui ont conduit à la création d'un programme évoluent. Un programme doit donc pouvoir être modifié après la mise en production initiale. Cela concerne aussi bien la documentation du code en vue de la reprise par d'autres programmeurs que la modification ou l'ajout de fonctionnalités.
- **la réutilisabilité.** Le développement logiciel est une activité très chronophage. Il faut le plus possible essayer de réutiliser ce qui a déjà été développé dans des projets précédents.
- **la portabilité.** Cela concerne la possibilité d'exécuter un programme sur plusieurs types d'architectures matérielles et logicielles différentes. Ce point nous concerne dans son extension aux systèmes multi-agents distribués dans lesquels les agents se trouvant sur différentes architectures doivent pouvoir communiquer en utilisant un langage commun.
- **la fiabilité.** Ce point est certainement un des plus importants. Le développement logiciel est un processus extrêmement complexe soumis à une quantité faramineuse d'erreurs possibles. Pour éviter les bugs, il faut fournir aux programmeurs des outils les contraignants à rester dans des chemins balisés où le risque d'erreur est plus faible.

L'amélioration de la mise en œuvre de ces principes dans les systèmes s'est faite par l'introduction d'une succession de paradigmes : des modèles cohérents de vision de la modélisation des applications.

Le paradigme objet est le plus couramment utilisé actuellement. L'encapsulation des données permet un faible couplage entre les entités du système et une réutilisabilité dans différents contextes. L'héritage et le polymorphisme permettent de simplifier le développement d'entités légèrement différentes de celles déjà créées en réutilisant du code déjà éprouvé.

Le paradigme agent, évolution des travaux initiés par Carl Hewitt sur les langages d'acteur [Hewitt *et al.*, 1973], semble correspondre à une évolution majeure des techniques de modélisation. Il peut être vu comme un remplaçant avantageux du paradigme objet. Le concept d'autonomie doit permettre, en découplant les entités du système de manière beaucoup plus drastique qu'auparavant, de casser la complexité structurelle des systèmes. Les agents devraient par exemple être capables de continuer à fonctionner si leurs liens de communication avec les autres agents sont coupés, ou si un agent ne répond pas (ou pas favorablement) à une de leurs requêtes.

Dans beaucoup de systèmes multi-agents, le concept d'agent est une abstraction qui n'est utilisée que pendant la phase de modélisation. Et ce sont les techniques objet classiques qui sont utilisées

pour l'implémentation. Nous pensons au contraire que, comme le paradigme agent utilise un haut niveau d'abstraction, les développeurs devraient avoir à leur disposition des outils également de haut niveau qui leur permettent de s'abstraire des difficultés de plus bas niveau. Le fossé réduit entre la modélisation et l'implémentation est certainement une des caractéristiques qui fait le succès d'un paradigme, comme c'est le cas avec l'objet.

1.3 La combinaison des deux domaines pour améliorer la gestion du temps dans les SMA

Nous aimerions pouvoir utiliser dans nos systèmes multi-agents le plus grand nombre possible de résultats des recherches en IA qui améliorent la gestion du temps. Nous aimerions également pouvoir apporter aux systèmes issus de l'IA des outils de niveau méta permettant d'améliorer leur comportement temporel.

Les principes du génie logiciel entrent cependant parfois en conflit avec les méthodes imaginées pour résoudre des problèmes, en particulier les problèmes très complexes traités par l'IA[2]. Si on ne prend pas garde aux problèmes de mise en œuvre des techniques que nous utilisons et proposons, on risque de se retrouver avec un système trop complexe pour pouvoir être utilisé facilement et sans risque d'apparition de dysfonctionnements non prévus.

Le temps apparaît de manière transverse et sous de multiples formes dans les systèmes multi-agents cognitifs. Contrairement aux objets, qui subissent le temps, les agents cognitifs sont pro-actifs. Ils ont la responsabilité de décider quand exécuter les actions qui leur permettent d'atteindre leurs buts. Ils ont ainsi besoin d'un module pour planifier des actions, gérer leurs dépendances, ... Un autre module peut être chargé d'ordonnancer les différentes tâches. La planification et l'ordonnancement sont deux problèmes difficiles auxquels on se confronte dans la plupart des applications multi-agent. Lorsque les actions planifiées s'inscrivent dans la durée, il faut mettre en place un dernier module, trop souvent négligé, chargé de l'exécution des actions. Ce module doit être capable de démarrer des actions, de suivre leur déroulement, de lire leur état et modifier leurs paramètres pendant l'exécution, de vérifier leur bonne terminaison et éventuellement de les arrêter si l'agent le décide.

Le fonctionnement d'un agent peut être cadencé par une horloge, par l'arrivée d'événements extérieurs ou les deux à la fois. Lorsque c'est une horloge qui donne le rythme, il convient d'être certain que l'ensemble des traitements nécessaires à un cycle pourront être effectués avant le prochain top d'horloge. Ce point peut poser problème notamment dans les simulations où l'on veut pouvoir compresser le temps en jouant sur la rapidité d'avancement de l'horloge virtuelle utilisée pour cadencer la simulation. D'un autre côté, on associe généralement la notion de réactivité à l'arrivée d'événements extérieurs. L'idée est de gérer au mieux la file d'attente des messages entrants en fonction du temps

[2] Qui ne s'est jamais exclamé « C'est une usine à gaz ! » en lisant la description d'une architecture complexe ? Même en ayant une vision très élitiste de la programmation, il paraît clair qu'un programmeur ne doit jamais se sentir dépassé par le système qu'il utilise.

de latence maximal que l'on s'autorise avant qu'un événement ne soit traité. Il convient toujours de trouver le bon compromis entre réactivité et cognition : un agent qui est en train de concevoir ou d'exécuter un plan doit rester capable de prendre en compte les événements extérieurs, quitte à devoir remettre en cause ce qu'il avait déjà planifié. Les travaux sur les protocoles de communication entre agents ont permis de mieux structurer la manière de prendre en compte l'événement particulier qu'est l'arrivée d'un message. Notons que le comportement d'un agent pro-actif ne peut pas être exclusivement dirigé par l'arrivée de messages. En effet, si ses interlocuteurs décident de ne pas envoyer les messages requis ou qu'ils sont perdus, l'agent resterait bloqué en attente d'un message qui n'arrivera jamais. Il convient de systématiquement spécifier le comportement de l'agent dans ce cas. On met généralement en place un système de réveil qui réactive l'agent au bout d'un certain temps. Ce même mécanisme de réveil peut être utilisé par les autres modules de l'agent qui ont besoin d'être activés à des dates données.

Le temps peut également apparaître dans la base de connaissances de l'agent, en datant chaque information ou en conservant différents historiques (croyances, actions, événements, . . .). La base de connaissances est utilisée pour effectuer des raisonnements, qui peuvent porter ou être conditionnés par le temps.

Le temps est aussi une composante importante dans la vérification des systèmes, puisque l'on veut pouvoir vérifier qu'un agent respectera bien ses spécifications temporelles et qu'aucune condition temporelle ne pourra affecter son fonctionnement.

Enfin, lorsque l'on conçoit un système multi-agent qui doit s'exécuter sur un ordinateur, on se confronte à des aspects beaucoup plus système de la gestion du temps comme la gestion et le partage des ressources processeur disponibles. À ce niveau, la difficulté principale provient de la confrontation entre les niveaux d'abstraction agent et système. Au niveau système, on dispose des notions de processus ou processus léger (threads), de priorité, d'ordonnancement de processus avec respect de délais ou non, de temps partagé, . . . Au niveau agent, on parlera plutôt d'autonomie, de pro-activité, de réactivité, de contextes de communication ou de raisonnement, . . . Les solutions proposées au niveau du système d'exploitation pour gérer le partage des ressources processeur entrent en conflit avec les propriétés recherchées au niveau agent. Par exemple, la notion de priorité gérée de manière centralisée au niveau de l'ordonnanceur du système limite considérablement l'autonomie des agents. De plus, le système ne dispose pas des informations qui pourraient être utiles pour prendre de bonnes décisions. En effet, le système se trouve à un niveau d'abstraction inférieur au niveau agent et n'a pas accès aux connaissances des agents sur leurs besoins en ressources processeur, sur les délais qu'ils doivent respecter, sur la marge de manœuvre dont ils disposent, sur leurs interactions, . . . Cela peut également poser des limites par rapport aux autres propriétés des agents.

1.4 Objectifs

Nous avons remarqué dans la section précédente que la gestion du temps est multi-forme. Nous nous intéressons dans cette thèse uniquement à l'aspect système : la gestion et le partage des ressources

processeur entre les différents agents. Nous avons en effet choisi d'utiliser une approche ascendante pour pouvoir prendre en compte petit à petit les différentes facettes de la gestion du temps. Nous avons mis en place dans cette thèse les mécanismes les plus bas niveau pour faciliter la gestion du temps dans nos systèmes multi-agents. Cette étape nous permet de réduire le fossé qui existe entre les niveaux d'abstraction du système d'exploitation et des systèmes multi-agents. Nous disposons ainsi de bases solides sur lesquelles nous pourrons à l'avenir ajouter d'autres couches chargées de gérer les autres aspects liés au temps. Dans cette démarche, nous voulons utiliser au maximum, sans les modifier, les fonctionnalités proposées par les systèmes d'exploitation. Cela concerne notamment la partie ordonnancement : si l'ordonnanceur du système d'exploitation ne dispose pas de toutes les propriétés nécessaires à nos systèmes multi-agents, nous fournirons les fonctionnalités manquantes sous forme d'une bibliothèque de notre plateforme multi-agents.

Du point de vue génie logiciel, la question à laquelle nous voulons répondre est la suivante : comment programmer des agents utilisant des techniques de l'IA pour bénéficier des nouveaux principes liés à ce paradigme comme le couplage très faible entre les agents tout en conservant ou améliorant les bénéfices liés aux précédents principes (simplicité, productivité, efficacité, réutilisabilité, . . .) ?

Nous donnons une importance très grande aux principes de simplicité et de fiabilité. Nous n'hésiterons donc pas à contraindre très fortement les programmeurs pour que la programmation d'agents gérant plusieurs contextes simultanément ne soit pas un casse-tête et que l'apparition d'interblocages (*deadlocks*) soit très limitée. Nous nous attacherons également à ce que les propriétés que nous attribuons aux agents restent visibles au niveau de la programmation. Ce sera notamment le cas de l'autonomie, qu'il est très facile de perdre au niveau de l'implémentation, mais qui est un des piliers du découplage des entités du système.

Nous considérerons dans la suite que nos agents sont des agents logiciels dont les actions dans l'environnement peuvent être ramenées à des interactions avec les autres agents par envoi de messages et à des tâches de calcul. Les calculs peuvent être de toutes sortes, mais nous nous préoccupons principalement de ceux réalisés par les algorithmes de l'Intelligence Artificielle.

Nous nous intéressons plus particulièrement dans cette thèse aux aspects suivants :

- La gestion simultanée de plusieurs contextes au sein d'un agent. Cela concerne les contextes de dialogue avec les autres agents ainsi que les contextes d'exécution des actions planifiées de l'agent.
- Les actions que réalisent nos agents étant des calculs potentiellement longs, nous nous intéressons à la représentation de ceux-ci et aux raisonnements qu'il est possible de réaliser sur la représentation choisie.
- La gestion des signaux d'attention permettant une réactivité du système par rapport aux événements extérieurs.
- Les contraintes temporelles que les agents peuvent avoir à respecter.
- Le partage des ressources de calcul d'un ou plusieurs processeurs entre les agents, qu'ils aient des contraintes temporelles ou non.

1.5 Organisation du document

Ce document est organisé en 7 chapitres. Le premier est le chapitre d'introduction qui présente d'une part l'objectif et le cadre général de la thèse et d'autre part la structuration choisie pour ce document.

Le problème de la gestion du temps, et plus particulièrement celui du partage des ressources processeur, fait l'objet du chapitre 2. La présentation des travaux connexes est organisée en trois parties : la section 2.2.1 traite des difficultés qui apparaissent dès que l'on utilise un algorithme. Les solutions classiques pour gérer simultanément plusieurs algorithmes et les problèmes qu'elles apportent sont détaillées dans la section 2.2.2. La section 2.2.3 présente les langages et techniques proposés pour introduire le temps dans les raisonnements, les comportements et l'interaction des systèmes intelligents avec leur environnement.

Le chapitre 3 décrit la solution que je propose pour que les agents puissent gérer en parallèle plusieurs contextes de calcul tout en restant à l'écoute du monde extérieur. Cette proposition repose sur le concept d'artifact initialement développé par une équipe italienne pour la coordination [Omicini *et al.*, 2004]. Les artifacts peuvent être vus comme des outils que les agents utilisent pour atteindre leurs buts. Ils disposent de modes d'emploi qui permettent aux agents de comprendre comment les utiliser. Je décris dans ce chapitre les extensions que j'ai apportées pour adapter les artifacts à l'externalisation des calculs longs. Je propose un ensemble de modes d'emploi d'artifacts de calcul à utiliser en fonction des propriétés temporelles de l'algorithme encapsulé dans l'artifact.

Le chapitre 4 traite plus particulièrement du partage efficace des ressources processeur entre les entités du système multi-agent. Ce problème est étudié selon que les agents ont des délais à respecter pour l'utilisation des ressources processeur (section 4.1) ou qu'ils n'en ont pas (section 4.2). Pour résoudre ce problème, je propose d'utiliser un artifact de coordination qui centralise les demandes de ressources processeur. Je fournis un algorithme d'ordonnancement qui respecte les contraintes spécifiques à nos systèmes.

Le chapitre 5 décrit la mise en œuvre des artifacts de calcul et des techniques de partage des ressources processeur proposées dans la plateforme multi-agents développée au sein de Thales. Les expérimentations que j'ai menées sur la plateforme sont présentées. Elles concernent l'utilisation de notre artifact de coordination dans une application existante développée par Thales. Je montre que cela permet d'améliorer le respect des échéances. Je montre également comment notre artifact de coordination peut être utilisé en mode multi-machine pour équilibrer la charge entre les différents processeurs disponibles.

Enfin, les chapitres 6 et 7 concluent sur les résultats de cette étude et indiquent quelles sont les nouvelles pistes à aborder par la suite dans les trois directions suivantes : gestion du temps dans les systèmes multi-agents, l'évolution du modèle d'agent et la manière avec laquelle les agents manipulent les artifacts de calcul.

Chapitre 2

Contexte

Concevez toujours une chose en la considérant dans un contexte plus large — une chaise dans une pièce, une pièce dans une maison, une maison dans un quartier, un quartier dans une ville.
Eliel Saarinen.

Nous décrivons dans ce chapitre les problèmes que pose la gestion du temps dans les systèmes multi-agents. L'exposé des principaux résultats de la théorie de la complexité nous permettra de définir de manière plus formelle quelles sont les difficultés auxquelles nous avons à faire face lorsque l'on s'intéresse à la résolution d'un unique problème. Nous verrons quelles sont les techniques qui ont été proposées pour contourner ces difficultés. Dans un second temps, nous verrons quelles approches et techniques ont été proposées pour bénéficier des capacités de distribution des calculs des systèmes informatiques actuels. Nous verrons enfin quelles sont les solutions actuellement proposées pour que des agents puissent gérer plusieurs contextes d'exécution simultanés.

2.1 Exposé du problème

2.1.1 Des agents autonomes, extravertis et conscients du temps

Le paradigme agent apparaît comme un moyen de réduire la complexité structurelle des systèmes. En concevant une application comme un ensemble d'entités indépendantes lors de l'exécution, on réduit le couplage entre les différents éléments du système et donc la programmation des entités du système se fait de manière plus indépendante également. Nous avons remarqué que la gestion du temps apparaît de manière transverse aux différentes fonctionnalités des agents intelligents.

Nous visons des applications dans lesquelles les agents ont des calculs longs à réaliser (principalement des calculs issus d'algorithmes de l'Intelligence Artificielle) et peuvent avoir des contraintes

temporelles à respecter pour ces tâches de calcul. Le paradigme agent repose sur des principes comme l'autonomie ou la réaction rapide aux changements de l'environnement. Si l'on perd ces propriétés, on aura perdu une bonne partie de l'intérêt du paradigme agent. Nous nous attacherons donc à ce que ces propriétés soient conservées dans nos systèmes, quitte à faire des sacrifices sur d'autres aspects.

La question du sens précis que nous donnons à l'autonomie dans nos SMA sera traitée en profondeur dans la section 3.1.3 lorsque nous aurons clairement défini l'environnement dans lequel s'exécutent nos agents. Nous pouvons pour l'instant nous contenter du sens général communément admis : un agent est considéré comme autonome s'il est capable d'agir seul en fonction des informations lui provenant de l'environnement. Un agent ne peut être autonome que s'il est capable de pro-activité, c'est-à-dire qu'il peut avoir un comportement piloté par des buts et ainsi prendre l'initiative des actions qu'il exécute.

Nous voulons également que nos agents aient un comportement extraverti. Nous considérerons dans la suite que les agents interagissent avec l'environnement uniquement par envoi et réception de messages. Chaque agent dispose d'une boîte aux lettres dans laquelle s'empilent les messages qu'il n'a pas encore pris en compte. Nous utilisons la définition suivante pour la propriété d'extraversion :

Définition 2.1 (Agent extraverti/agent introverti) *Un agent est extraverti si le temps processeur qui sépare deux consultations de sa boîte aux lettres est borné. Il est introverti dans le cas contraire.*

Cette définition implique qu'un agent extraverti ne peut s'engager aveuglément dans des tâches de calcul longues qui l'empêcheraient de consulter régulièrement sa boîte aux lettres.

Nous voulons enfin que nos agents soient conscient du temps. Nous définissons cette propriété ainsi :

Définition 2.2 (Agent conscient du temps) *Un agent est conscient du temps si ses raisonnements et donc son comportement dépendent du temps qui s'écoule.*

Il ne suffit pas qu'un agent raisonne sur le temps pour qu'il soit conscient du temps qui passe. Il faut de plus que ses raisonnements prennent en compte l'écoulement du temps pendant ses actions et prises de décision. La connaissance de l'écoulement du temps s'acquiert en consultant l'heure auprès d'une horloge. L'horloge à laquelle un agent conscient du temps se réfère n'a que peu d'importance. En particulier, il n'est pas nécessaire qu'elle soit la même pour les différents agents conscients du temps d'un même SMA. Remarquons cependant que si des agents conscients du temps doivent partager des informations temporelles (pour se coordonner par exemple), ils vont devoir se synchroniser sur une horloge commune. Nous utiliserons d'ailleurs une horloge commune à l'ensemble des agents se trouvant sur une machine dans le chapitre 4.

Un agent conscient du temps pourra planifier des tâches de calcul, estimer la durée nécessaire à leur exécution, les lancer, les surveiller et prendre des décisions en fonction de ses observations. Un

agent doit par exemple pouvoir détecter qu'un traitement dure anormalement longtemps et décider de le stopper définitivement. Il doit pouvoir continuer de fonctionner sans les données qu'il attendait en sortie d'un traitement s'il décide de le stopper en cours d'exécution.

Nous avons décidé d'adopter dans cette étude une approche ascendante dans laquelle nous proposons des outils génériques assez bas niveau et indépendants du modèle d'agent utilisé. Notre priorité est d'obtenir des agents ayant les trois propriétés décrites précédemment (autonomie, extraversion et conscience du temps) tout en étant capable de réaliser des calculs complexes potentiellement longs.

2.1.2 Le problème des longs calculs

2.1.2.1 Utilisation d'algorithmes complexes

Nous voulons faciliter l'intégration de tous types d'algorithmes dans les systèmes multi-agents, en particulier ceux issus du domaine de l'Intelligence Artificielle. Ces algorithmes peuvent avoir des temps d'exécution longs, ainsi que de mauvaises propriétés temporelles. Par mauvaises propriétés temporelles, nous entendons que les algorithmes utilisés peuvent avoir des durées d'exécution hautement variables en fonction des données en entrée et que ces durées sont difficiles à prévoir. Il existe des techniques pour palier ces difficultés. Nous les détaillons dans la section 2.2.1. Par exemple, on peut tenter de trouver un algorithme incrémental pour notre problème. Celui-ci sera plus facilement intégrable dans un SMA qu'un algorithme non incrémental. Cependant, les algorithmes de l'IA sont la plupart du temps assez complexes et il n'est pas toujours possible d'en améliorer les propriétés temporelles facilement. Nous voulons donc pouvoir intégrer dans nos SMA les algorithmes issus de l'IA tels quels.

L'exécution par les agents d'algorithmes ayant des durées d'exécution longues va à l'encontre du principe d'extraversion défini plus haut. Les agents deviennent plutôt introvertis : ils se remettent en phase avec leur environnement une fois leurs calculs terminés. En se comportant de cette manière, un agent risque par exemple de continuer un long calcul malgré le fait qu'un autre agent lui ait envoyé un message lui indiquant que le calcul est devenu inutile. La durée d'exécution d'un algorithme étant une notion relative, nous considérerons qu'un algorithme a une durée d'exécution longue lorsqu'elle dépasse la durée séparant deux événements qu'un agent doit prendre en compte : pour conserver un comportement extraverti à l'agent, le programmeur ne peut plus dans ce cas négliger le temps pris par l'exécution de l'algorithme considéré. Il devra disposer d'un moyen pour prendre en compte le nouvel événement, tout en laissant l'algorithme continuer son exécution.

2.1.2.2 Utilisation de code hérité

Un paradigme tel que celui d'agent ne peut être facilement adopté qu'à la condition que l'on puisse réutiliser la base de code dont on dispose dans de nouvelles applications ou dans la refonte

d’une application avec le paradigme agent. La capacité de pouvoir « agentifier » du code existant est un élément essentiel à l’utilisation du paradigme agent dans des applications industrielles.

Le fait de considérer l’utilisation de code hérité n’est cependant pas qu’une problématique industrielle car nous avons vu dans le paragraphe précédent que la plupart du temps il n’est pas envisageable de modifier l’algorithme dont on dispose pour qu’il ait de meilleures propriétés temporelles. Il faut donc se résoudre à devoir intégrer dans nos systèmes des boîtes noires pour lesquelles on ne peut toucher au code, soit parce qu’on ne peut le faire (code hérité), soit parce qu’on se refuse à le faire (algorithme complexe).

L’architecture du système multi-agent doit donc être suffisamment flexible pour introduire du code non agent. Elle doit également apporter des outils pour supporter l’exécution de tous types d’algorithmes sans que leur mauvaises propriétés nuisent à l’intégrité du système.

Le concept de réutilisabilité peut également être vu non pas du point de vue du programmeur, mais du point de vue agent : les agents devraient pouvoir dynamiquement apprendre à utiliser de nouveaux outils de calcul disponibles dans le système multi-agent pour atteindre leurs buts. Pour cela, les outils de calcul devraient disposer d’un mode d’emploi qui indique aux agents comment les utiliser.

2.1.3 Le problème du partage des ressources processeur

2.1.3.1 Partage d’un processeur

Les systèmes multi-agents sont par définition des systèmes distribués mais il est très souvent inconcevable de considérer que chaque agent dispose de son propre processeur et qu’il sera libre d’utiliser toutes les ressources disponibles pour son seul usage personnel. Il arrive d’ailleurs très fréquemment que l’ensemble des agents d’un SMA s’exécutent sur un unique processeur.

Nous devons donc nous résoudre à ce que nos agents partagent les ressources de calcul d’un unique processeur. Nous devrons fournir des outils pour que les agents puissent respecter les délais qu’ils peuvent avoir sur leurs tâches de calcul, tout en conservant au maximum leur autonomie. Il faudra également veiller à un autre point très important : si les agents partagent un unique « cerveau », pendant le temps où un agent l’utilise, les autres « perdent conscience ». Nous devons donc fournir aux concepteurs de SMA un niveau d’abstraction auquel les agents pourront disposer de la propriété de conscience du temps dont nous parlions dans la section 2.1.1.

Nous trouvons également très important que les applications multi-agents que nous développons puissent s’exécuter sur des machines de bureau classiques et donc sur les systèmes d’exploitation classiquement installés sur ce type de machines (Windows, Unix). Il ne paraît pas concevable de devoir installer un système d’exploitation particulier — temps réel par exemple — pour que nos applications fonctionnent. L’architecture de nos SMA devra donc s’adapter à ce type de système, en tentant d’utiliser au mieux les fonctionnalités de gestion des processus proposés. S’il manque des fonctionnalités (comme la gestion de délais sur les tâches, voir section 4.1), elles devront être apportées par les bibliothèques de nos SMA sans avoir à toucher au système d’exploitation lui-même.

2.1.3.2 Partage de plusieurs processeurs

Nous venons d'indiquer que nous devons fournir des outils pour que nos agents puissent partager un unique processeur. Il ne faut cependant en aucun cas se limiter à l'utilisation d'un unique processeur si plusieurs autres sont disponibles. Nous devons également fournir des outils permettant de répartir les agents sur différentes machines. La politique de répartition effective doit être laissée au concepteur de chaque application, mais nous pouvons fournir des outils aidant à la mise en place de telle ou telle politique de répartition. Nous pouvons par exemple fournir :
- des moyens pour lancer un agent sur une machine distante ou de faire migrer un agent déjà démarré sur une autre machine,
- des moyens pour mesurer la charge d'un processeur et avertir un agent qu'il pourrait disposer de plus de ressources de calcul sur une autre machine.

2.1.4 Modèle des applications visées

Nous donnons dans cette section un modèle d'application qui résume les objectifs précédemment cités. Nous donnons également la liste des contraintes supplémentaires que nous avons utilisées dans le cadre de cette étude.

Nous voulons implémenter ou réimplémenter des applications avec le paradigme agent. Il faudra permettre l'intégration de code hérité dans les systèmes multi-agents. Les actions que les agents ont a réaliser peuvent être réduites à des tâches de calcul.

Les agents ne partagent pas de mémoire et ne communiquent entre eux que par échange de messages. Ils doivent être capables de gérer différents contextes simultanément. Ceux-ci peuvent être de deux types : contexte de dialogue avec un autre agent ou contexte d'exécution d'un calcul. Les tâches de calcul doivent pouvoir encapsuler du code hérité aux propriétés temporelles plus ou moins bonnes et l'agent doit être capable de bénéficier au mieux des propriétés temporelles des algorithmes dont il dispose pour réaliser ses tâches de calcul.

Nos systèmes multi-agents doivent fonctionner sur les systèmes d'exploitation couramment utilisés sur les machines de bureau. Nous ne voulons pas développer de système large échelle et nous considérons qu'une limite d'une dizaine d'agents s'exécutant sur un unique processeur est raisonnable. Nos applications fonctionnent principalement sur une unique machine, mais nous sommes à la recherche d'un système permettant de bénéficier de la puissance de calcul de plusieurs processeurs de manière assez simple.

Les agents peuvent avoir des contraintes temporelles à respecter. Celles-ci s'expriment par le respect de délais pour les tâches de calcul. Les contraintes temporelles sont molles, c'est-à-dire qu'elles seront satisfaites au plus près des dates limites, mais pourront les dépasser légèrement en fonction de la politique du système d'exploitation sous-jacent, qui ne garantit pas le respect de contraintes temporelles dures. Un dépassement maximal de l'ordre de la seconde est autorisé dans nos systèmes.

Les contraintes temporelles sont attachées à des tâches de calcul généralement longues et de toute façon de durées supérieures à la seconde. La gestion du respect des contraintes temporelles doit être indépendante de la manière avec laquelle l'agent détermine ses besoins en ressources processeur.

Nous considérons dans la suite qu'il n'y a pas d'interactions directes entre les différents algorithmes mis en œuvre par les agents : nous ne nous intéressons pas aux cas où les différents algorithmes auraient besoin de données calculées par les autres et donc auraient besoin d'interagir directement. Les seuls cas qui nous intéressent sont les suivants :

– les agents réalisent des tâches de calcul indépendantes,
– les agents réalisent des tâches de calcul qui dépendent les unes des autres, mais les dépendances sont gérées au niveau agent dans le plan de l'agent ou au niveau du SMA grâce aux interactions entre les agents,
– les agents utilisent différentes heuristiques pour résoudre un unique problème, mais les heuristiques fonctionnent indépendamment les unes des autres.

2.2 Travaux connexes

2.2.1 La gestion du temps d'exécution des algorithmes

2.2.1.1 Algorithmes

Le mot algorithme a été introduit par le moine Abelard de Bath par référence au nom du mathématicien perse Abou Jafar Muhammad Ibn Musa al-Khuwarizmi qui, au neuvième siècle écrivit le premier ouvrage systématique sur la solution des équations linéaires et quadratiques. Rétro-activement, on considère que c'est la méthode d'Euclide pour déterminer le plus grand commun diviseur d'entiers naturels qui est le premier algorithme a avoir été conçu.

De manière générale, un algorithme est un énoncé, dans un langage bien défini, d'une suite d'opérations permettant de résoudre un problème par calcul.

Les travaux des mathématiciens de la première moitié du siècle dernier ont permis de préciser et de formaliser la notion d'algorithme en ne considérant que les méthodes effectivement implantables sur une machine. Le premier résultat important est plutôt décevant puisqu'il nous indique qu'il existe des fonctions qui ne sont pas calculables. C'est le cas notamment avec le problème de l'arrêt : Turing a démontré qu'il n'existe pas de programme universel qui prenne n'importe quel programme en argument et qui, en temps fini, renvoie « oui » si l'exécution du programme reçu en argument finit par s'arrêter et « non » s'il ne finit pas.

Un programme informatique peut être vu comme la réalisation d'un algorithme sur une machine donnée. On considère que c'est la comtesse Augusta Ada Lovelace qui aurait écrit ou participé à l'écriture du premier programme informatique en 1843. Celui-ci permettait de calculer les nombres de Bernoulli avec la machine analytique que Charles Babbage avait entrepris de construire.

L'autre résultat important a été proposé de manière indépendante par Turing et Church. La thèse de Church-Turing, dans sa formulation la plus courante, indique que les machines de Turing formalisent correctement la notion de *méthode effective de calcul*. Et donc que toute fonction calculable l'est sur une machine de Turing qui « termine ». Cette thèse ne peut être démontrée, mais elle est communément admise comme vraie, en particulier parce que Church et Turing y sont parvenus par des chemins totalement différents.

Nous pouvons maintenant donner une définition plus précise d'un algorithme.

Définition 2.3 (Algorithme (de [Knuth, 1997]) *Un algorithme est un ensemble fini de règles qui donnent la séquence d'opérations pour résoudre un type de problème particulier. Un algorithme doit posséder toutes les caractéristiques suivantes :*
- *un algorithme doit se terminer après un nombre fini d'étapes,*
- *les étapes de l'algorithme doivent être définies précisément, le mieux étant d'utiliser un langage de programmation pour les exprimer,*
- *un algorithme doit prendre 0 ou plus valeurs en entrée,*
- *un algorithme possède une ou plusieurs valeurs de sortie qui sont en relation avec les valeurs en entrée,*
- *l'algorithme doit être effectif, c'est-à-dire que les opérations qui le composent sont suffisamment simple pour pouvoir être exécutées de manière atomique.*

On peut classer les algorithmes en fonction du type de résultat attendu. La classe la plus remarquable est celle des problèmes de décision puisque c'est elle qui a permis a Turing d'aboutir à son résultat sur le problème de l'arrêt. Elle est importante également parce que tout les problèmes peuvent se ramener à un problème de décision. Par exemple, le problème de la k-colorabilité d'un graphe G peut être formulé ainsi :« existe-t-il une coloration à k couleurs du graphe G ? » où k et G sont tout les deux des paramètres du problème.

2.2.1.2 Classes de complexité

Pour résoudre un problème avec un ordinateur, il ne suffit pas de trouver un algorithme car les algorithmes eux-même posent problème !

Un algorithme aura beau être concis et élégant, il ne sera d'aucune utilité s'il n'est pas possible de le faire fonctionner en pratique sur les architectures matérielles dont on dispose. Cette impossibilité d'utilisation peut notamment apparaître si l'algorithme utilise trop de mémoire ou s'il s'exécute dans des temps irraisonnable. C'est pourquoi depuis les années 1970, une étude systématique des propriétés des algorithmes a été menée. Elle a aboutie à une classification des problèmes et de leurs algorithmes associés en fonction de leur difficulté.

Tout d'abord, nous pouvons classer les algorithmes en fonction de la quantité de mémoire ou du nombre d'instructions dont ils vont avoir besoin pour s'exécuter. Ces deux quantités dépendent de la

taille des données en entrée. De plus, il est la plupart du temps inutile de comparer les performances des algorithmes sur des tailles de données petites. C'est essentiellement le comportement asymptotique qui nous intéressera. Pour cela, on utilise couramment la notation dite de Landau, en réalité introduite un peu plus tôt par Bachmann.

On écrit $f(x) = O(g(x))$ si et seulement s'il existe des constances C et N telles que :

$$\forall x > N, |f(x)| \leq C|g(x)|$$

Ce qui signifie intuitivement que f ne croît pas plus vite que g.

La classification que nous obtenons est la suivante.

notation	**complexité**
$O(1)$	constante
$O(log(n))$	logarithmique
$O(nlog(n))$	parfois appelée « linéarithmique »
$O((log(n))^c)$	polylogarithmique
$O(n)$	linéaire
$O(n^2)$	quadratique
$O(n^c)$	polynomiale, parfois « géometrique »
$O(c^n)$	exponentielle
$O(n!)$	factorielle

Les algorithmes en $O(1)$ ont un temps d'exécution qui est indépendant de la taille des données. Ils ne sont pas très courants, mais on peut notamment noter que l'algorithme d'ordonnancement intégré dans le noyau de Linux est maintenant en $O(1)$, ce qui lui permet de rester efficace sur des machines massivement parallèles. Les algorithmes logarithmiques ou linéaires sont considérés comme facilement exploitables. Les difficultés commencent à apparaître lorsque les algorithmes sont quadratiques ou polynomiaux. Et les algorithmes exponentiels ou factoriels sont impraticables en dehors des instances de problèmes jouets ne contenant que quelques données.

La complexité peut être calculée dans le meilleur des cas — ce qui a peu d'intérêt —, en moyenne ou au pire des cas. La complexité au pire des cas permet d'obtenir une borne supérieure qui peut être utilisée quand on conçoit des systèmes temps réels durs. L'ordonnancement est alors effectué en prenant en compte le temps d'exécution au pire des cas (WCET : Worst Case Execution Time en anglais). Ces techniques sont impraticables lorsque l'on utilise des algorithmes aux complexités au pire des cas très élevées. C'est d'autant plus dommageable que la plupart du temps, la complexité en moyenne est beaucoup plus limitée que la complexité au pire des cas. On doit dans ce cas plutôt utiliser comme base de calcul la complexité moyenne. On renonce ainsi à respecter des contraintes temps réelles dures.

La théorie de la complexité donne une classification complémentaire de la première. Elle différencie les algorithmes déterministes et les algorithmes non déterministes. Alors qu'un algorithme déterministe produit un seul calcul à chaque étape, un algorithme non déterministe produit un ensemble de calcul.

La classification suivante permet de déterminer la difficulté d'un problème :

Classe	**Description**
L	les problèmes de décision qui peuvent être résolus par un algorithme déterministe en espace logarithmique par rapport à la taille de l'instance
NL	cette classe correspond à la précédente mais pour un algorithme non-déterministe
P	les problèmes de décision qui peuvent être décidés par un algorithme déterministe en un temps polynomial par rapport à la taille de l'instance
NP	les problèmes de décision pour lesquels la réponse « oui » peut être décidée par un algorithme non-déterministe en un temps polynomial par rapport à la taille de l'instance
PSPACE	les problèmes décidables par un algorithme déterministe en espace polynomial par rapport à la taille de son instance
NPSPACE	les problèmes décidables par un algorithme non-déterministe en espace polynomial par rapport à la taille de son instance
EXPTIME	les problèmes décidables par un algorithme déterministe en temps exponentiel par rapport à la taille de son instance

Nous connaissons un certain nombre de relations entre ces ensembles. Elles sont résumées ici :

$$L \subset NL \subset P \subset NP \subset PSPACE = NPSPACE$$

Le problème toujours irrésolu actuellement est de déterminer si P=NP. Pour l'instant, on considère que ce n'est pas le cas. Nous verrons dans les sections suivantes que beaucoup d'efforts sont faits pour pouvoir trouver des solutions au problèmes de la classe NP. Cependant, notons que ce ne sont pas les seuls problèmes qui posent des difficultés de gestion du temps. D'un certain point de vue, la gestion par un agent de l'exécution de calculs polynomiaux sur de très grands ensembles de données n'est pas très différente de l'utilisation d'un algorithme exponentiel sur un problème de petite taille. Dans les deux cas, la durée d'exécution peut être longue et difficile à estimer (de manière précise sur les instances des données en entrée). Nous ne limiterons pas notre étude de l'encapsulation des tâches de calcul aux problèmes difficiles. Les problèmes plus faciles ont également de l'intérêt, par exemple pour contrôler de manière intelligente des calculs scientifiques très volumineux.

2.2.1.3 Utilisation d'heuristiques

Si un problème peut être exprimé sous la forme de la recherche d'un élément maximisant ou minimisant une fonction dans un grand espace de recherche, la seule méthode sûre pour trouver le bon élément consiste à parcourir de manière exhaustive l'espace de recherche. Ceci est inconcevable lorsque l'espace de recherche est très grand.

On essaye alors, en utilisant les propriétés de l'espace de recherche spécifiques au problème posé, de trouver l'élément recherché en se rapprochant petit à petit de lui à partir d'un élément initial. Si l'heuristique est bien choisie, la complexité moyenne peut éventuellement être dans une classe inférieure (par exemple, polynômiale au lieu d'exponentielle) à celle de l'algorithme qui explorerait l'ensemble dans un ordre inapproprié.

Cheeseman *et al.* ont essayé de donner une méthode pour déterminer « où se trouvent les problèmes vraiment durs » [Cheeseman *et al.*, 1991]. Ils ont découvert que des problèmes comme la K-colorabilité d'un graphe ou le voyageur de commerce possèdent des *paramètres d'ordre* et que les instances dures de ces problèmes se trouvent autour de certaines valeurs critiques de ces paramètres. Ils espéraient ainsi ouvrir une nouvelle voie où l'on pourrait produire des « diagrammes de phase » que l'on utiliserait pour prédire la difficulté de résolution d'une instance donnée des problèmes. Cette technique n'a vraisemblablement pas été mise en oeuvre avec d'autres types de problèmes pratiques.

La détermination d'une heuristique tient donc encore beaucoup de règles déterminées empiriquement en étudiant les propriétés statistiques du problème sur des instances réalisables en pratique. On a pu cependant trouver quelques modèles de conception d'heuristiques, notamment pour les problèmes d'optimisation. C'est ce qu'on appelle des métaheuristiques. Les plus connues sont le recuit simulé, la recherche tabou et les algorithmes génétiques.

Les mêmes métaheuristiques peuvent être utilisées pour résoudre des problèmes d'optimisation multi-objectifs. Dans ce dernier cas, on recherche un ensemble de solutions non dominées appelé le « front de Pareto ». Ce sont les solutions parmi lesquelles on ne peut décider si une solution est meilleure qu'une autre, aucune n'étant systématiquement inférieure aux autres sur tous les objectifs.

La tendance actuelle est à l'utilisation conjointe de plusieurs métaheuristiques collaborant à la résolution du problème. On peut par exemple alterner l'exploration de l'espace de recherche grâce à un algorithme génétique et l'exploitation grâce à un recuit simulé. Matthieu Basseur a récemment proposé ce type de fonctionnement coopératif pour des problèmes d'ordonnancement de type flow-shop [Basseur, 2005]. Il nous paraît intéressant de voir ce type de problèmes dans une perspective multi-agent où chaque méthode serait gérée par un agent et la coopération entre les agents permettrait une alternance entre les méthodes intelligemment guidée. Les techniques proposées dans cette thèse constituent un premier pas dans cette direction.

Remarquons que l'utilisation d'heuristiques nous oblige la plupart du temps à changer l'énoncé du problème pour ne considérer qu'une restriction du problème initial. Par exemple, pour le problème du voyageur de commerce, le problème initial consiste à trouver *le circuit Hamiltonien le plus court* dans un graphe dont les nœuds représentent des villes et les arcs sont annotés par la distance entre les deux villes qu'ils relient. Ce problème ne peut être résolu en pratique pour des problèmes de grande taille. On décide donc de changer l'énoncé du problème pour par exemple tenter de trouver *un chemin le plus court possible*.

Cette tendance à résoudre mieux, ou de manière satisfaisante, des problèmes restreints ne doit pas être perçue comme un aveu d'échec. D'ailleurs, elle prend tout son sens depuis les travaux de Herbert

Simon sur la notion de *satisficing* et la tendance a vouloir construire des agents rationnels plutôt que des agents « parfaits ».

2.2.1.4 Algorithmique anytime

L'utilisation d'algorithmes exhaustifs ou d'heuristiques pose le problème de la boîte noire qui prend la main et ne la rend que quand elle a terminé. L'idée serait de rendre la boîte un peu transparente pour pouvoir voir de l'extérieur l'état d'avancement du calcul et d'autoriser quelques fonctions de contrôle comme la pause, la reprise et l'arrêt avec renvoi de la valeur courante. C'est ce que tente de réaliser l'algorithmique anytime (figure 2.1).

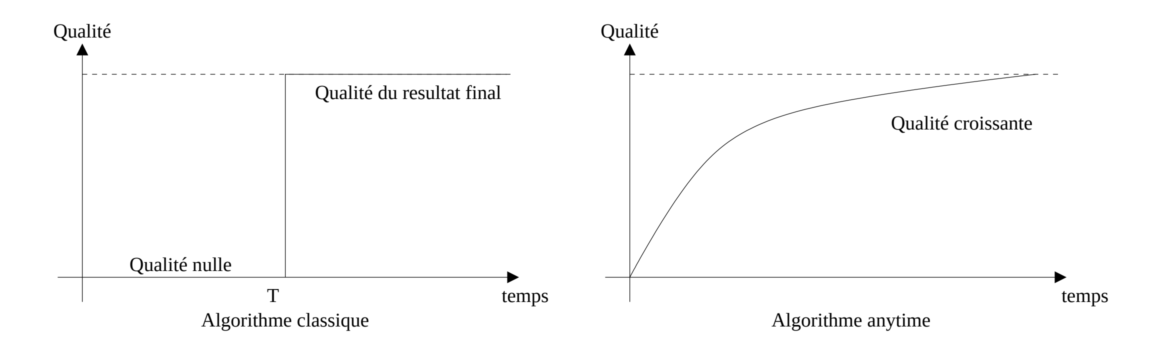


FIG. 2.1 – Différence entre un algorithme classique et un algorithme anytime.

Dean et Boddy donnent la définition suivante [Dean et Boddy, 1988, Boddy et Dean, 1989] :

Définition 2.4 (Algorithme anytime) *Un algorithme anytime est un algorithme itératif qui garantit de produire une réponse à toute étape de calcul, où la réponse est supposée s'améliorer à chaque itération. Du point de vue de l'implantation, un algorithme anytime doit posséder les caractéristiques suivantes :*
- interruptibilité : *il peut être interrompu à tout moment,*
- préemptibilité : *il doit pouvoir être suspendu et relancé au même point avec un temps de réponse négligeable afin de se prêter aux techniques d'ordonnancement.*

Le terme anytime est souvent utilisé de manière abusive pour décrire un algorithme seulement interruptible et/ou incrémental. Nous prendrons dans la suite une définition plus stricte de la notion d'algorithme anytime dans laquelle l'algorithme considéré dispose obligatoirement d'une capacité de prédiction de la qualité de la solution en fonction du temps de calcul qui lui a été alloué.

La prédiction de la qualité est réalisée au moyen d'un profil de performance qui montre l'évolution de la qualité de la solution en fonction du temps. Un agent décide du temps d'exécution d'un algorithme en mettant en perspective la qualité espérée du résultat et l'utilité correspondante. En effet, il arrive souvent que l'utilité d'une donnée décroisse en fonction du temps.

Nous retenons les définitions de qualité et d'utilité données dans [Delhay, 2000] :

Définition 2.5 (Qualité) *Pour un algorithme donné, la qualité du résultat Q(t,d) de l'algorithme est une application de l'ensemble des couples (temps de calcul t, donnée en entrée d) dans* $\mathbb{R}$ *ou [0,1].*

Définition 2.6 (Utilité) *Pour un algorithme donné, l'utilité U(q,t) d'un résultat est une application de l'ensemble des couples (temps de calcul, qualité) dans* $\mathbb{R}$ *ou [0,1].*

L'algorithmique anytime semble résoudre bon nombre des problèmes posés par les algorithmes lorsqu'ils sont mis en pratique. Il y a cependant deux problèmes qui empêchent d'utiliser de manière systématique des algorithmes anytime :

– trouver un algorithme incrémental et interruptible pour un problème donné n'est pas toujours très aisé,
– et lorsque l'on a trouvé l'algorithme, c'est la détermination du critère de qualité et du profil de performance qui pose problème [Delhay, 2000].

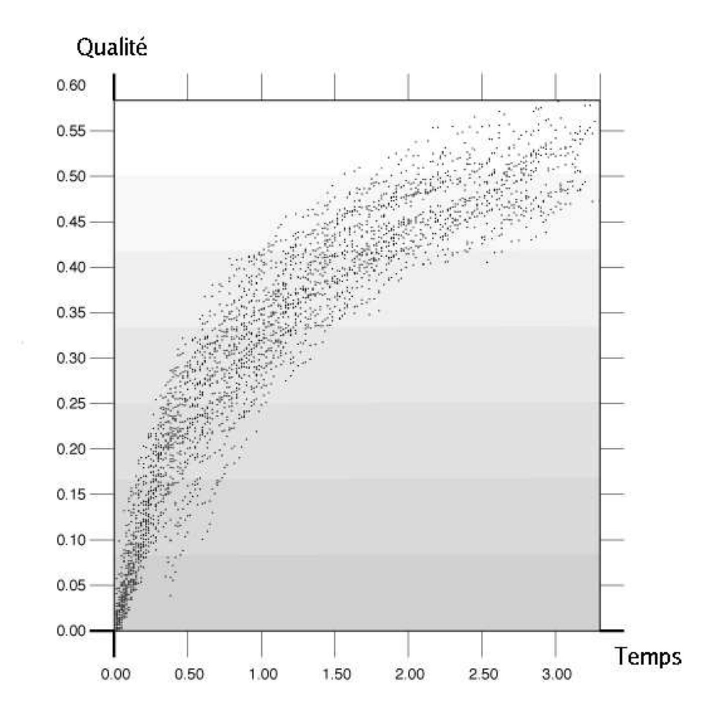


FIG. 2.2 – Qualité de l'algorithme du voyageur de commerce construit avec la méthode statistique [Grass, 1996].

La figure 2.2 nous permet d'illustrer les difficultés posées par ce dernier point. Elle montre la carte de qualité obtenue en utilisant une méthode statistique sur le problème du voyageur de commerce. Le critère de qualité est le rapport du coût du chemin optimal sur le coût du chemin courant.

Le problème provient du choix des données en entrée utilisées pour cette phase d'apprentissage des performances de l'algorithme. [Delhay, 2000] montre que les méthodes statistiques sont rarement la bonne méthode pour obtenir un profil de performance qui prédisent efficacement un comportement réaliste de l'algorithme.

Mouaddib et Zilberstein ont proposé le *raisonnement progressif* comme une alternative aux algorithmes anytime [Zilberstein et Mouaddib, 1999]. Le raisonnement progressif est utilisé dans les cas où il est difficile de déterminer un profil de performance précis. Il utilise plusieurs niveaux de traitements afin de transformer graduellement une solution imprécise en solution précise.

Nous verrons dans la suite que, même si on n'utilise pas souvent dans la pratique des algorithmes anytime, beaucoup d'idées précédemment citées sont réutilisées dans les architectures d'agent proposées.

2.2.2 La distribution des calculs : approches et techniques

L'adage « diviser pour régner », généralement vu de manière négative, prend un sens intéressant lorsqu'il est appliqué à l'informatique. L'idée de résoudre un problème en le décomposant en sous-parties permet de s'occuper de chaque sous-problème individuellement sans se préoccuper de la difficulté du problème initial. On peut ainsi faire résoudre les sous-problèmes par différents fils d'exécution. Il paraît aussi indispensable de pouvoir faire résoudre plusieurs problèmes en même temps à un ordinateur.

Nous présentons maintenant les différentes approches et techniques associées qui ont été proposées pour distribuer des calculs. La présentation est découpée en fonction du type de support matériel de la distribution. Nous mettons en évidence les avantages et les inconvénients du point de vue génie logiciel et du point de vue de l'utilisateur du système. Cela nous permettra de déterminer dans les chapitres suivants comment nous voulons implanter le parallélisme à l'intérieur de nos agents et aussi entre eux.

2.2.2.1 Cas d'une machine

Il n'est pas nécessaires de disposer de plusieurs processeurs pour voir des avantages à la séparation de ce qui est exécuté sur le processeur en plusieurs contextes. Cette séparation permet :

– d'exécuter plusieurs programmes indépendants, par exemple pour être capable de gérer plusieurs utilisateurs simultanément,
– d'exécuter des fils d'exécution qui dépendent les uns des autres et sont capables d'accéder aux mêmes zones de mémoire.

On peut ainsi découper un programme en plusieurs sous-programmes qui s'exécutent de manière concurrente. Les fils d'exécution à programmer contiennent moins de code mais il faut faire attention à la complexité importante apportée par les mécanismes de synchronisation qu'il faut mettre en place. Nous détaillons maintenant les techniques proposées pour réaliser ce parallélisme et la synchronisation associée.

2.2.2.1.1 Le multi-tâche

La manière la plus simple pour gérer plusieurs programmes simultanément consiste à demander aux programmeurs d'indiquer au système d'exploitation à quel moment donner la main à un autre programme. Il dispose pour cela d'un appel système `yield()`. Cette méthode, le multi-tâche coopératif, a l'avantage d'être simple à mettre en œuvre. Elle permet de donner deux niveaux d'illusion à l'utilisateur :

– Un utilisateur n'a pas à être conscient que d'autres utilisateurs utilisent le système en même temps que lui. En effet, si les ressources processeur disponibles pour un utilisateur donné lui suffisent, il ne s'aperçoit pas que d'autres utilisateurs consomment des ressources en même temps que lui. Les utilisateurs peuvent travailler de manière indépendante sans avoir à se préoccuper des autres utilisateurs.
– Un utilisateur peut voir plusieurs de ses programmes s'exécuter simultanément si le passage de l'un à l'autre est suffisamment fréquent.

Elle possède cependant de nombreux inconvénients visibles au niveau des programmeurs. C'est une forme de couplage fort dans laquelle le développeur d'un programme ne peut faire abstraction du fait qu'il y aura d'autres programmes qui vont s'exécuter sur le processeur. La robustesse est aussi limitée car le système entier peut s'arrêter si un des processus n'exécute jamais l'instruction `yield()`.

La méthode actuellement utilisée sur la plupart des systèmes d'exploitation disponibles se nomme multi-tâche préemptif. L'appel système `yield()` n'est plus utilisé et c'est le système d'exploitation qui décide quand attribuer le processeur à un autre programme. Cette méthode permet, tout comme le multi-tâche coopératif, de donner à l'utilisateur l'illusion de plusieurs programmes s'exécutant en même temps. Il est important de noter que cette illusion est maintenant propagée jusqu'à la programmation des différents programmes : pendant la phase de programmation, on n'a plus à se soucier du fait qu'il y aura d'autres programmes qui vont s'exécuter en même temps que celui qu'on est en train de créer. Le couplage entre les programmes est ainsi très fortement diminué et la robustesse est accrue car un processus ne peut plus bloquer toute la machine et empêcher les autres de s'exécuter. L'illusion de programmes s'exécutant en parallèle est obtenue en attribuant à chaque processus de tout petits quanta de temps. Le passage régulier d'un processus à l'autre permet d'assurer une certaine réactivité à chacun des programmes.

Le multi-tâche préemptif peut être utilisé pour paralléliser l'exécution de plusieurs programmes indépendants, mais aussi pour paralléliser l'exécution d'un seul programme. On utilise pour cela la technologie des processus légers (*threads*). Chaque processus léger est un fil d'exécution qui peut s'exécuter en parallèle des autres, tout comme avec les processus normaux. La différence se situe au niveau de l'accès à la mémoire : deux processus ne partagent pas de mémoire tandis que tous les processus légers qui constituent un processus partagent la même mémoire.

Cette possibilité de pouvoir partager de la mémoire entre les différents fils d'exécution est très intéressante mais elle pose un nombre incroyable de difficultés de programmation. [Lee, 2006] effectue le constat que la programmation avec les processus légers fait perdre des propriétés essentielles de la

programmation séquentielle : compréhensibilité, comportement prédictible et déterminisme. La programmation parallèle revient donc en grande partie à tenter de supprimer l'indéterminisme apporté par l'introduction des processus légers. C'est une tâche fastidieuse, même sur des cas simples. Lee montre sur l'exemple, pourtant minimaliste, du modèle de conception *observateur* [Gamma *et al.*, 1993] les difficultés et subtilités auxquelles on est confrontés. Il détaille les raisonnements qu'il convient de mener pour passer d'une implémentation valide pour un processus léger à une implémentation valide avec plusieurs processus légers. Bien que l'exemple est donné sur un modèle de conception très simple, les raisonnements ne sont pas triviaux et mettent parfois en jeu 3 ou 4 processus légers pour faire apparaître une situation d'interblocage.

La programmation multi-tâche n'est pas aisée mais elle apporte l'illusion de programmes indépendants s'exécutant en parallèle, bien qu'au final il n'y ait qu'un seul processeur. Nous pensons que ce point est très important dans notre cadre. Quand nous concevons nos agents, nous voulons nous placer à ce niveau d'abstraction pour pouvoir considérer que les agents s'exécutent en parallèle sur le processeur. C'est à cette condition qu'on peut les voir comme constamment conscients et qu'on peut les programmer de manière indépendante. Cependant, nous voulons que les programmeurs aient le moins de difficultés possible pour construire de tels systèmes. Nous nous attacherons dans la suite à ce que les outils proposés soient simples à comprendre et que leur utilisation soit sans surprise, c'est-à-dire qu'ils devront limiter au maximum l'indéterminisme inhérent à la programmation multi-tâche.

2.2.2.1.2 Les systèmes d'exploitation temps réel

Lorsque l'on parle de gestion du temps dans les systèmes informatiques, on en vient très rapidement à évoquer les systèmes d'exploitation temps réel. Un système est dit temps réel s'il est capable de réaliser les traitements qu'on lui demande à un rythme adapté aux données d'entrée qui lui parviennent.

Un système est temps réel dur si toutes les tâches doivent se terminer avant une date donnée sans aucun dépassement de délai. Il est temps réel mou s'il supporte que les délais soient légèrement dépassés de temps en temps. Les systèmes de traitement de flux vidéo sont par exemple souvent considérés comme des systèmes temps réel mous car si le traitement d'une image prend trop de temps, la suivante est simplement sautée pour rattraper le retard. Ce comportement est acceptable si la fréquence des images sautées n'est pas trop importante.

La gestion du respect des dates limites assignées aux tâches est dévolue à un ordonnanceur. Ils peuvent être classés de différentes manières. Tout d'abord, ils peuvent fonctionner en ligne (*online*) ou hors ligne (*offline*). On utilise un algorithme hors ligne lorsque l'ensemble des processus qui vont tourner sur le système est fixe et que les propriétés des traitements effectués par les processus sont également fixes. Ainsi, en sachant que l'on aura n processus à gérer avec pour chaque processus i un besoin périodique de période P_i de la quantité W_i de ressources processeur, l'ordonnanceur dispose de toutes les informations nécessaires pour calculer (et optimiser) un ordonnancement qui convient. À l'exécution, l'ordonnanceur ne fait qu'utiliser les informations précalculées pour déterminer quand

lancer ou stopper les processus. Ce type de technique est utilisé dans les systèmes extrèmement critiques car on garantit hors ligne qu'il n'y aura pas de « mauvaise surprise » à l'exécution : on prouve que le système respectera les délais pour toutes les tâches.

Lorsque le nombre de processus à l'exécution est variable ou que l'on ne connaît pas à l'avance tous les paramètres liés aux tâches, on utilise un ordonnanceur en ligne, c'est-à-dire un ordonnanceur qui recalcule à l'exécution un nouvel ordonnancement à chaque fois qu'un paramètre est modifié.

Les contraintes temporelles sont fournies à l'ordonnanceur en spécifiant l'exécution temporelle des actions. Une action peut être :

- **périodique** si elle doit être exécutée à un intervalle donné. La date limite pour chaque exécution peut être différente de la période. Ce type d'action est par exemple utilisé lorsque les traitements doivent respecter la fréquence d'échantillonage de données captées.
- **apériodique** si le temps entre chaque nouvelle exécution est aléatoire, par exemple quand l'action correspond au traitement d'un événement.
- **sporadique** si elle est exécutée à un intervalle donné, mais avec une variation (*jitter*).

Comme on veut garantir que les délais seront respectés, on utilise le temps d'exécution au pire des cas (*WCET* : *Worst Case Execution Time* en anglais) comme donnée d'utilisation du processeur pour chacune des tâches de calcul.

Pour pouvoir être mis en œuvre, un algorithme en ligne doit disposer au moins de deux propriétés : il doit être correct et complet. Il doit donner un résultat correct et trouver un ordonnancement lorsqu'il en existe un, ceci pour tous les jeux de données d'entrée possibles. Liu et Layland ont présenté en 1973 plusieurs algorithmes facilement implémentables qui ont ces propriétés, dont les algorithmes *rate monotonic* (monotonique par taux) et *Earliest Deadline First* (première échéance d'abord).

Rate monotonic (RM) ne sait gérer que des tâches périodiques. Le processus élu à un instant donné est celui dont l'inverse de la période est le plus fort. C'est un algorithme préemptif et optimal dans la classe des algorithmes à priorités fixes. Si l'on considère que les échéances sont fixées à la fin de chaque période, Liu et Layland ont donné une condition suffisante pour que RM produise un ordonnancement qui respecte toutes les échéances : il suffit que le taux d'occupation processeur pour l'ensemble des tâches ne dépasse pas 69,3%. Si l'on dépasse ce taux d'occupation, on n'est plus certain d'obtenir un ordonnancement valide. C'est une condition nécessaire mais non suffisante : il peut exister des ordonnancements valides avec un taux d'occupation supérieur à 69,3%. Si par contre les échéances sont inférieures aux périodes, il n'existe plus de condition suffisante basée sur la charge pour être certain d'obtenir un ordonnancement valide.

Earliest Deadline First (EDF) permet de gérer des tâches périodiques et apériodiques. Il exécute à un instant donné la tâche dont la date limite est la plus proche, ceci jusqu'à sa terminaison complète. Il supprime de la liste des tâches celle qui vient de se terminer, détermine celle dont la date limite est maintenant la plus proche et débute son exécution. EDF permet de respecter toutes les échéances jusqu'à un taux d'occupation de 100% et il est optimal dans la classe des algorithmes à priorités dynamiques.

Less Laxity First (LLF) est une variante d'EDF qui dispose des mêmes propriétés. On favorise ici les tâches urgentes en exécutant à un instant donné la tâche dont la marge (échéance - durée d'exécution restant) est la plus faible.

Il est possible de lancer sur un système temps réel des processus qui n'ont pas de contraintes temporelles. On fait pour cela apparaître la notion de priorité entre les deux classes de processus. Les processus temps réel ont une priorité stricte par rapport aux processus non temps réel. Ainsi, les seconds ne peuvent s'exécuter que dans les « trous » laissés par les premiers dans l'ordonnancement.

Lorsque le système est en surcharge, ce sont tout d'abord les processus non temps réel qui ne peuvent plus s'exécuter. Si cela ne suffit pas, il faut accepter que certaines échéances ne soient pas respectées. Rate-monotonic est plus intéressant qu'EDF en cas de surcharge car ce dernier va pénaliser les autres tâches pour favoriser la tâche qui va manquer son échéance. On peut aussi mettre en place un système de priorité entre les processus temps réel pour garantir que certains pourront toujours s'exécuter même en cas de surcharge. Il convient cependant de remarquer que dans ce cas, la synchronisation entre les tâches par utilisation de sémaphore peut faire apparaître un problème connu sous le nom d'inversion de priorité. En effet, une tâche de plus faible priorité peut bloquer une tâche de plus haute priorité si elle accède en section critique à une ressource partagée.

Cet exposé des principes de fonctionnement des systèmes temps réels permet de se rendre compte que ceux-ci ne peuvent être utilisés que dans un cadre bien particulier où l'on est capable de fournir à l'ordonnanceur toutes les informations nécessaires à ses calculs. L'obligation de fournir le temps d'exécution au pire des cas pour garantir le respect des contraintes temporelles est une limitation importante pour l'utilisation de systèmes temps réel pour exécuter des systèmes intelligents utilisant des algorithmes complexes. En effet, ces derniers peuvent avoir des temps d'exécution au pire des cas extrèmement élevés par rapport au temps d'exécution en moyenne. De plus, les systèmes temps réel ne sont pas tous préemptifs et sont incapables de fournir l'illusion que les processus sont actifs en permanence. Enfin, les solutions proposées en cas de surcharge du système sont des solutions très autoritaires où ce sont toujours les mêmes processus qui sont évincés sans même en être avertis. Cela n'est pas satisfaisant lorsque les applications mettent en jeu des agents intelligents autonomes.

2.2.2.1.3 Les processeurs multi-cœurs

Nous mentionnons ici les processeurs multi-cœurs car ceux-ci vont devenir de plus en plus prépondérants à l'avenir dans les machines de bureau. Nous en sommes actuellement à deux ou quatre cœurs par processeur et les fondeurs comme Intel parlent de 40, 80 voire 100 cœurs par processeur d'ici une dizaine d'années.

Leur architecture permet d'exécuter en parallèle au moins un processus léger sur chacun des cœurs. En théorie, il n'y a pas besoin de modifier les applications pour bénéficier du parallélisme apporté par ces processeurs. En pratique, les applications actuelles n'utilisent pour la plupart les processus légers que pour assurer la réactivité de l'interface graphique pendant que les calculs longs sont

réalisés par un seul d'entre eux. Les gains n'apparaissent donc pour l'instant que lorsque l'on utilise en même temps plusieurs applications gourmandes en temps processeur. Pour bénéficier pleinement de la puissance de ces processeurs à l'avenir, il faudra revoir l'architecture de toutes les applications pour que les calculs longs soient réalisés par un nombre variable de processus légers en fonction du nombre de cœurs disponibles sur la machine. Or, nous venons de voir que la programmation des processus légers est sujette à de nombreuses difficultés. Les fondeurs sont conscients de ces difficultés et proposent des bibliothèques qui simplifient la programmation des processus légers. C'est le cas d'Intel, avec sa bibliothèque « Threading Building Blocks ». Elle fournit des instructions pour paralléliser les boucles (`for` et `while`), des structures de données sûres pour une utilisation concurrente (vecteur, file, table de hachage) et des primitives de synchronisation pour les cas d'utilisation usuels.

Il y a également la possibilité d'utiliser des bibliothèques de plus haut niveau déjà développées pour bénéficier du parallélisme des machines multiprocesseurs et des grappes de serveurs. Nous les décrivons dans le paragraphe suivant.

À l'heure actuelle, ces processeurs n'autorisent pas un contrôle fin de l'allocation des processus aux différents cœurs au niveau de l'utilisateur : il est impossible d'indiquer à un processus de s'exécuter sur un cœur particulier, à moins de modifier l'ordonnanceur du système d'exploitation. Cette dernière possibilité n'est pas raisonnable la plupart du temps. L'utilisation de processeurs multi-cœurs dans notre cadre, où nous voulons pouvoir raisonner sur les ressources processeur attribuées à chaque agent, est donc assez problématique. Espérons qu'à l'avenir, nous disposerons sur ces processeurs de possibilités de contrôle au niveau utilisateur.

2.2.2.1.4 Les machines multiprocesseurs et les systèmes de gestion de grappes de serveurs

La mise au point de machines multiprocesseurs ou massivement multiprocesseurs se fait souvent de manière conjointe aux niveaux matériel et logiciel. La préoccupation principale est le partage efficace de la mémoire pour obtenir des gains de performance les plus proches possibles des gains théoriques. En effet, en théorie, avec n processeurs, on doit pouvoir obtenir un résultat n fois plus vite qu'avec un unique processeur. En pratique, on n'obtient pas cette accélération à cause des temps de communication et la synchronisation nécessaire entre les différents processeurs. Ces aspects, développés de manière spécifique dans [Culler *et al.*, 1999], ne nous intéressent pas ici. Nous nous intéressons plutôt aux bibliothèques logicielles fournies avec les architectures matérielles.

Les machines multiprocesseurs sont souvent très onéreuses. Pour limiter les coûts, on peut regrouper plusieurs ordinateurs indépendants appelés nœuds au sein d'une grappe de serveurs. Les grappes sont gérées par un logiciel de *clustering* qui permet à l'utilisateur de ne voir qu'une unique machine multiprocesseur à mémoire partagée. Il est chargé de gérer le cycle de vie des processus : placement des processus sur les nœuds, passage des données à traiter et récupération des résultats. Il permet également de gérer les défaillances matérielles en redémarrant les processus d'un nœud défaillant sur un autre nœud.

Ces systèmes sont très intéressants car ils ne requièrent aucune modification des applications et que la vue proposée à l'utilisateur est très simple. Cependant, les logiciels de clustering actuellement proposés ne sont pas très flexibles. Ils tentent uniquement de faire de l'équilibrage de charge pour que tous les processeurs soient utilisés de la même manière et sont incapables de gérer des ordonnancements plus complexes requis par exemple quand on veut donner des priorités aux tâches ou que l'on a des délais à respecter pour certaines tâches et pas pour d'autres. Les logiciels de clustering sont donc plutôt handicapants pour nos systèmes dans lesquels les agents devraient pouvoir décider eux-même du processeur sur lequel ils vont s'exécuter. Nous ne pouvons pas encore utiliser nos systèmes multi-agents sur des grappes de serveurs, mais nous aurons dans l'avenir la possibilité de le faire. En effet, les grappes sont gérées de manière logicielle. Certains projets sont même accessibles de manière totalement libres comme openMosix [Buytaert, 2005], qui est conçu comme une extension du noyau Linux pour permettre d'obtenir une image unique du système d'exploitation sur un réseau de machines (*Single-System Image* ou *SSI* en anglais). Nous pouvons dès à présent planifier dans nos travaux futurs l'élaboration d'un logiciel hybride gestion de grappe/système multi-agent permettant de combiner les avantages des deux approches : simplicité, efficacité et robustesse des grappes d'une part ; et gestion intelligente des ressources processeur adaptée aux systèmes multi-agents d'autre part.

Nous présentons dans la suite les deux bibliothèques les plus connues pour programmer des programmes parallèles sur les machines multiprocesseurs et les grappes de serveurs. Ce sont des bibliothèques standards qui sont portées sur la plupart des architectures.

MPI

MPI signifie Message Passing Interface. C'est une bibliothèque dans laquelle la mémoire est distribuée et la communication entre les processus se fait par échange de messages de manière explicite dans les programmes. Les deux primitives de base sont `MPI_Send()` et `MPI_Recv()`. Elles permettent respectivement d'envoyer un message à un ou plusieurs destinataires et d'attendre la réception d'un message.

Cette structuration autour des messages échangés entre les entités est très proche de nos systèmes multi-agents où les agents ne communiquent entre eux que par messages et ne partagent pas de données. En ce sens, MPI pourrait être vu comme un système simple mais robuste de gestion de système multi-agent gérant le cycle de vie des agents, le placement sur différents processeurs et assurant le passage de messages entre les agents.

OpenMP

OpenMP [OpenMP,] propose un modèle de programmation à mémoire partagée. Il se présente sous la forme d'une bibliothèque C/C++ ou Fortran qui encapsule la gestion de processus légers et rend la communication entre les fils d'exécution implicite.

Un programme OpenMP classique commence son exécution de manière séquentielle. Cela correspond à la tâche maître, chargée de lancer une équipe de tâches filles qui s'exécutent toutes en même temps dans la région parallèle du programme. A la fin des calculs, toutes les tâches se synchronisent et seule la tâche maître continue son exécution. OpenMP propose des directives que l'on rajoute à un programme séquentiel pour le transformer en un programme parallèle. On peut ainsi très facilement paralléliser des boucles. L'exemple suivant montre comment utiliser `n` processus légers en parallèle pour parcourir une boucle `for`.

```
int i, c, d;
omp_set_num_threads(n);
#pragma omp parallel
#pragma omp for schedule(static, 5) private(c)
  for (i = 0; i < max; i++)
    {
      c = i;
      d += f(c);
    }
```

La variable `c` est privée à chacun des processus légers : il y aura une copie indépendante de `c` pour chaque processus léger. En revanche, la variable `d` est partagée par l'ensemble des processus légers. Les indices des itérations données à chaque processus léger sont déterminés selon la politique d'ordonnancement choisie. Ici, `schedule(static, 5)` indique que les 5 premières itérations seront données au premier processus léger, les 5 suivantes au second, ... Lorsque la quantité de calcul n'est pas identique pour chaque tour de boucle, il est possible d'indiquer un ordonnancement dynamique où un nouveau travail est donné à un processus léger quand il a terminé le précédent, sans s'occuper de l'état d'avancement des autres.

OpenMP propose également des primitives de synchronisation :

- `BARRIER` pour indiquer un point qui doit être atteint par tous les processus légers avant de pouvoir continuer,
- `SINGLE / END SINGLE` pour indiquer une section de code qui ne doit être exécutée que par un seul processus léger,
- `CRITICAL(name) / END CRITICAL(name)` pour encadrer les sections critiques dans lesquelles on ne peut avoir qu'un seul processus léger à la fois.

OpenMP constitue une bonne encapsulation des processus légers mais ne supprime pas totalement les problèmes d'interblocages. La philosophie, à l'opposé de celle de MPI, est difficle à rapprocher de ce que nous voulons pour nos systèmes. Nous préférons voir nos agents comme des entités qui démarrent des tâches de calcul et dialoguent avec elles plutôt que comme un programme unique dans lequel on fait apparaître du parallélisme quand le besoin s'en fait sentir.

2.2.2.2 Cas multi-machine et solutions réseau

Dans les sections précédentes, nous avons vu comment est géré le parallélisme sur une unique machine réelle ou virtuelle. Nous présentons maintenant les techniques utilisées pour gérer le paral-

lélisme lorsque l'on dispose d'un système réellement distribué. On ne se base plus ici sur la parallélisation d'un unique algorithme sur plusieurs processus ou processus légers mais sur les technologies réseau qui permettent d'accéder à des ressources distantes.

2.2.2.2.1 Objets distribués

Le paradigme objet est le plus utilisé actuellement pour modéliser toutes sortes d'applications. Il a d'abord été utilisé dans le cadre d'applications centralisées. On dispose maintenant d'intergiciels comme CORBA permettant de faire des appels de méthodes sur des objets distants. Lorsque l'on veut faire un appel de méthode à distance, on se connecte à l'intergiciel et on récupère une référence locale à l'objet distant. Ensuite, on effectue les appels de méthode directement sur l'objet local. Le but de ces intergiciels est de masquer totalement le fait que les appels sont effectués à distance : en dehors de l'initialisation, rien ne différencie un appel de méthode local ou distant. Ces outils sont très pratiques lorsque l'on veut continuer à programmer les applications comme si elles étaient centralisées, mais ils ont le défaut de ne plus autoriser de raisonnement sur l'aspect distribué de celles-ci.

2.2.2.2.2 Composants

Les composants logiciels sont essentiellement des objets distribués qui disposent de leur propre fil d'exécution (généralement un processus léger) et encapsulés dans une couche gérant différents services comme la sécurité ou les transactions.

Alors que les objets distribués sont principalement utilisés en mode synchrone, les composants fournissent des sources et des puits d'événements qui leur permettent de fonctionner en mode asynchrone. On se rapproche ainsi de la vision agent où l'on fonctionne quasi uniquement par échange de messages en mode asynchrone.

Les composants exhibent les services qu'ils fournissent mais également les services requis pour leur bon fonctionnement. Cela permet de concevoir une application en conectant les ports fournis par des composants aux ports requis par d'autres.

Nous utilisons dans la suite la notion d'artifact, qui se rapproche de la notion de composant logiciel. La section 3.6 donne d'autres informations concernant les composants et compare en détail notre approche avec l'approche composant.

2.2.2.2.3 Le GRID

Alors qu'une grappe est un ensemble de machines homogènes regroupées dans un même lieu, une grille de calcul est un ensemble de ressources de calcul hétérogènes délocalisées. On peut dire que le GRID est aux systèmes multi-agents actuels ce que l'Internet est aux réseaux locaux : il y a un

différence d'échelle très importante. La communauté agent ayant beaucoup de problèmes de passage à l'échelle des architectures d'agent et des protocoles d'interaction, l'étude du modèle GRID paraît importante [Foster *et al.*, 2004]. La fusion des deux modèles est encore prématurée mais quelques études comme celle menée à l'université de Montpellier ([Jonquet *et al.*, 2006]) vont dans ce sens, en se focalisant sur la notion de service qui est commune aux approches agent et GRID.

Open Grid Services Architecture

Open Grid Services Architecture (OGSA) est l'architecture adoptée par de nombreux projets concernant le GRID. Les concepts d'OGSA sont dérivés des travaux présentés dans [Foster *et al.*, 2002]. L'infrastucture se base sur la technologie des Web Services et les technologies assciées comme WSDL et SOAP.

Dans OGSA, tout est décrit sous forme de service (une entité capable de communication réseau dont les compétences peuvent être utilisées par échanges de messages) : les ressources de calcul, les ressources de stockage, les gestionnaires de bases de données, les programmes, ... Cela permet de virtualiser tous les constituants de l'environnement GRID. En particulier, on parle d'organisations virtuelles.

Les fonctionnalités d'OGSA sont également fournies sous forme de services. On dispose ainsi de services de base pour l'authentification en mode *single sign on*, la tolérance aux fautes, la gestion du cycle de vie des services, ...

Le calcul distribué à la maison

Les ordinateurs de bureau étant largement sous-exploités, des projets scientifiques ayant de grands besoins en calculs proposent d'installer sur nos machines un programme qui va utiliser la puissance de milliers de machines pendant qu'elles sont inactives pour réaliser ces calculs. On peut citer dans les plus connus : Seti@home, Folding@Home, Décrypthon.

On parle de « desktop GRID ». Comme toute application GRID, les ressources de calcul sont hétérogènes, mais contrairement aux autres projets de GRID, ces systèmes sont ouverts et plutôt centralisés. L'ouverture à des clients pour lesquels on ne peut pas vérifier l'identité oblige à de la redondance : les calculs sont envoyés sur plusieurs clients puis leurs résultats sont comparés pour détecter un éventuel biais. Cela permet également de gérer la volatilité des clients.

La plupart de ces projets fonctionne en mode centralisé et il n'est pas possible que les serveurs centraux calculent eux-même l'ordonnancement des tâches sur les différents clients. Cette tâche est donc également déléguée aux machines clientes. Cela permet d'utiliser de nombreux critères dans l'ordonnancement comme le type de tâche, la puissance et la taille mémoire des clients, les bibliothèques de calcul scientifiques installées sur les clients, le clouage des données sur un client ou l'anticipation

des migrations, ... Ce dernier point est très important si on n'utilise pas de réseau dédié à large bande passante car le coût des communications devient plus grand que le coût des calculs.

Les projets de recherche actuels comme XtremWeb [Cappello *et al.*, 2005] développé au LRI à Paris tendent à rendre ces systèmes totalement distribués en se reposant sur les techniques des systèmes pair à pair. Dans ce type d'architecture, les applications qui proposent des calculs à réaliser sont elles-mêmes intégrées au système comme des pairs volontaires.

2.2.3 Gestion du temps dans les systèmes intelligents

Nous présentons dans cette section un tour d'horizon de la manière dont peut être géré le temps dans les systèmes intelligents. Les systèmes intelligents peuvent tout d'abord être conçus pour raisonner sur le temps et ceci indépendamment de l'exécution effective des actions décidées par l'agent. Les agents peuvent également avoir besoin de réagir rapidement aux événements qui se produisent dans leur environnement ou de gérer de manière efficace leurs conversations avec les autres agents. Nous verrons également comment les différents langages et architectures d'agents proposés gèrent le temps et la durée des actions que les agents entreprennent dans leur environnement.

2.2.3.1 Langages et techniques pour le raisonnement temporel

2.2.3.1.1 Langages logiques

Lorsque l'on veut raisonner sur le temps, on se tourne généralement vers une des nombreuses logiques temporelles proposées dans la littérature ([Fisher *et al.*, 2005]). Les logiques temporelles peuvent décrire des événements discrets, continus ou les deux.

Les logiques temporelles sont des logiques de propositions. Les propositions booléenne atomiques sont combinées par des connecteurs logiques (comme ET, OU, NON) et par d'autres opérateurs de modalité. Les logiques temporelles sont donc également des logiques modales.

On peut classer les logiques temporelles en différentes catégories. Tout d'abord, on peut définir des logiques linéaires comme Linear Temporal Logic (LTL). Elles permettent de décrire l'évolution d'un chemin temporel. Les opérateurs modaux que l'on peut utiliser sont les suivants :

- Xp : p sera vrai immédiatement après l'instant courant,
- Fp : p sera vrai un jour,
- Gp : p est toujours vrai,
- pUq : p doit être vrai jusqu'à ce que q devienne vrai,
- pRq : q est vrai jusqu'à ce que p devienne vrai.

On appelle ces opérateurs des opérateurs d'état.

Les logiques à branchement du temps (*branching-time temporal logics*) comme Computational Tree Logic (CTL) permettent quant à elles de décrire plusieurs chemins temporels se séparant à différents instants. On utilise les opérateurs décrits précédemment et on ajoute les deux opérateurs de chemin suivants :

- Ap : p est vrai dans tous les chemins temporels commençant dans l'état courant,
- Ep : il existe un chemin temporel commençant dans l'état courant dans lequel p est vrai.

Dans CTL, les opérateurs doivent être groupés par deux : un opérateur de chemin et un opérateur d'état. Cette contrainte est relachée dans CTL* : on peut mixer les opérateurs comme on veut. CTL et LTL sont des sous-ensembles de CTL*.

Plus récemment un nouveau type de logique temporelle a été définie par Alur, Henzinger et Kupferman : les Alternating-time Temporal Logics ([Alur *et al.*, 1998]). ATL est interprété sur des structures de jeux concurrents mettant en jeu un ensemble Σ de joueurs. On dispose d'un nouvel opérateur $\ll P \gg$, avec $P \subseteq \Sigma$. Ce nouvel opérateur généralise les opérateurs de chemin de CTL : $\ll \emptyset \gg$ correspond à l'opérateur A et $\ll \Sigma \gg$ correspond à l'opérateur E.

Enfin, on peut citer d'autres logiques basées sur les intervalles plutôt que sur des points. C'est le cas d'Interval Temporal Logic (ITL) proposé dans [Moszkowski, 1986]. Dans ITL, un intervalle est constitué d'une séquence potentiellement infinie d'états.

Les logiques temporelles sont principalement utilisées dans le cadre de la vérification formelle des systèmes. Elles ne peuvent généralement pas être utilisées pour programmer le comportement complet des agents.

2.2.3.1.2 Raisonnement sur des durées incertaines

De nombreux travaux ont porté sur l'incertain dans les raisonnements des agents, et en particulier sur les durées incertaines des calculs. Nous présentons ici les travaux dont la philosophie se rapproche le plus de la nôtre.

[Horvitz, 1990] et [Horvitz et Rutledge, 1991] traitent du problème de l'action dans l'incertain dans un cadre où l'utilité de l'action dépend du temps. Ils ont expérimenté leur programme Protos sur une application médicale simplifiée. Considérons le cas où le programme doit recommander un traitement pour un patient qui montre soudainement une pression sanguine extrèmement basse et de la tachycardie. Nous considérons que le problème a été réduit à deux syndromes exclusifs : défaillance cardiaque congestive (H1) et hypovolémie (H2). Bien que ces deux syndromes aient les mêmes symptomes, les traitements pour chacun d'eux entrent en conflit. Le traitement pour l'hypovolémie (A2) consiste à injecter au patient du liquide physiologique pour faire revenir le volume du sans à une valeur normale. Au contraire, le traitement pour la défaillance cardiaque congestive consiste à réduire la quantité de liquide dans le corps en administrant un diurétique. Bien évidemment, une erreur dans la décision de traitement à réaliser met la vie du patient en jeu. Protos prend la décision en fonction des différentes informations qui lui sont données : courbes d'utilité des actions en fonction du temps, seuils au dessous desquels il faut absolument déclencher une action, courbes de la pression du sang et du rythme cardiaque, . . .

2.2.3.2 Modèles d'agents hybrides

La dualité cognitif/réactif des agents a été remarquée dès le début des travaux de recherche sur les agents. En effet, un agent doit être capable d'effectuer des raisonnements à long terme tout en étant capable de prendre en compte *à temps* les changements dans l'environnement. La littérature propose de construire un agent en plusieurs couches. Les deux principaux modèles proposés sont InteRRaP [Muller et Pischel, 1993] et Touring Machines [Ferguson, 1992]. Nous les décrivons dans les paragraphes suivants.

2.2.3.2.1 InTeRRaP

Un agent InTeRRaP est constitué de trois couches[Muller et Pischel, 1993] : la première pour les comportements de type réflexe connus, la seconde pour les comportements nécessitant de la planification et la troisième pour les comportements mettant en œuvre la coopération d'autres agents. Lorsqu'un agent perçoit un changement dans l'environnement, une réaction est d'abord cherchée dans la première couche. Si aucune réaction n'est prévue dans cette couche, la seconde couche tente d'en trouver une en planifiant un ensemble de réactions de la première couche. En dernier recours, la main est donnée à la dernière couche. La figure 2.3 illustre cette organisation.

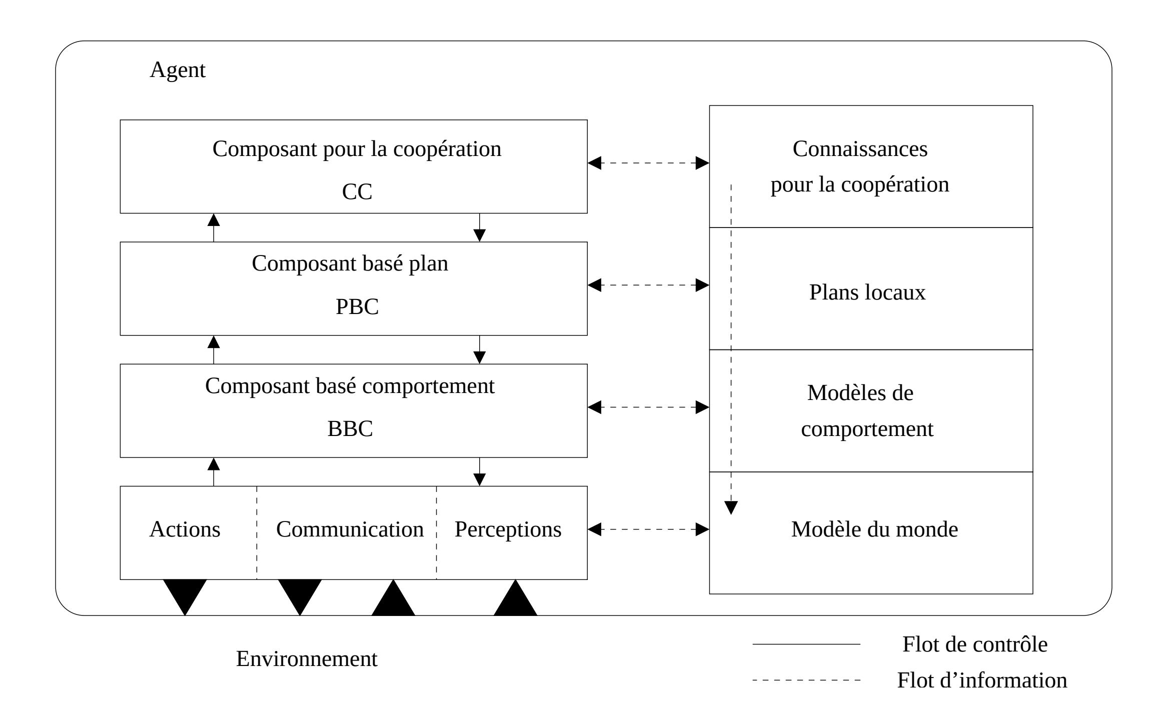


FIG. 2.3 – Architecture d'InteRRaP.

2.2.3.2.2 Touring Machines

Le modèle de Touring Machines [Ferguson, 1992] utilise lui aussi trois couches : pour les comportements réactifs, de planification et de résolution de conflits. Le comportement de l'agent est cyclique. A chaque top d'horloge, chaque couche propose une action en fonction des données fournies par les capteurs de l'agent (figure 2.4). Un module à base de règles est chargé d'inhiber les entrées ou les sorties de chacun des blocs pour qu'une seule action soit finalement sélectionnée.

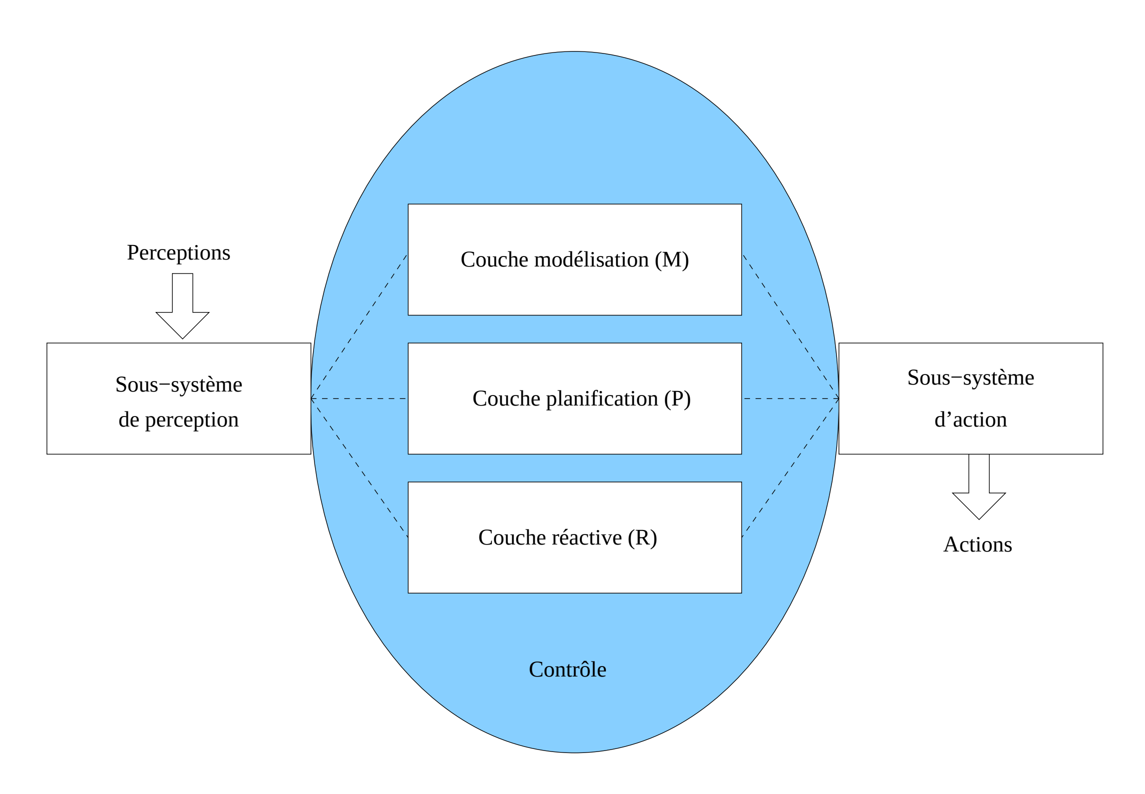


FIG. 2.4 – Architecture de Touring Machines.

2.2.3.3 Langages pour la gestion de contextes de communication

Nous voulons que nos agents aient un comportement extraverti. Il sera courant pour un agent d'être en conversation avec de multiples interlocuteurs simultanément. Il convient de structurer la gestion des communications autour des contextes de conversations entamées par les agents. Nous nous attardons donc quelque peu sur les langages proposés pour gérer ces différents contextes de communication au sein d'un agent.

2.2.3.3.1 COOL

COOL [Barbuceanu et Fox, 1995] est un langage permettant de spécifier, avec peu d'efforts de programmation, les mécanismes de coordination entre les agents. Chaque activité de coordination y est représentée par une machine à état finie.

Les *règles de conversation* sont la définition des transitions de l'automate : le type de message qui active la transition dans un état donné, les actions à réaliser lors du franchissement, les messages à envoyer et le nouvel état après franchissement. Les règles de conversation possèdent des variables locales instanciées permettant de transmettre des valeurs entre les transitions. Des constructions permettent de tester le premier message de la file d'attente de l'agent ou de chercher un message particulier dans la file. Des *règles de récupération après erreur* permettent de traiter les messages qui ne correspondent pas à ceux attendus dans un état donné. Elles sont séparées des règles de conversation pour pouvoir être réutilisées d'un type de conversation à l'autre.

La règle `r1` est un exemple de règle de conversation qui permet de passer de l'état `s0` à l'état `s1` si une proposition de produire quelque chose arrive et que l'agent est capable de la produire. Dans ce cas, un message d'acceptation est renvoyé.

```
(def-conversation-rule r1
  :current-state s0
  :received
    (propose :sender ?initiator :content (produce (?what))
  :such-that (achievable (produce (?what))
  :next-state s1
  :transmit (accept :content (produce (?what)))
```

Les *classes de conversation* sont les types d'automates que l'agent peut instancier. Une liste des instances de classes de conversations est conservée.

Exemple de classe de conversation :

```
(def-conversation-class conv-1
  :initiator: ?initiator
  :repsondent ?respondent
  :variables (?v1 ?v2)
  :initial state s0
  :final-state (s1)
  :conversaion-rules ((s0 r1) ...)
  :conversation-rule-applier CRA-1
  :error-rules (e1 e2 ...)
  :error-rule-applier ERA-1)
```

Des *règles de continuation* spécifient la manière dont les agents réagissent lors d'une requête de conversation venant d'un autre agent et la manière de choisir la prochaine conversation à suivre. Une seule conversation est suivie à tout instant. Pendant ce temps, les autres sont suspendues. Le programmeur peut ainsi spécifier la priorité à donner aux nouvelles conversations par rapport aux conversations déjà débutées. La règle suivante indique par exemple qu'une nouvelle requête de conversation

est servie s'il existe une classe de conversation qui accepte le premier message dans la file d'attente de l'agent.

```
(def-continuation-rule cont-1
  :input-queue-test
  (lambda(queue)
  (if queue (exists-conv-class-initially-accepting
       (first queue)) nil)))
```

Chaque instance de conversation possède un identifiant. COOL permet l'utilisation de multiples instances d'automates simultanément dans un agent. Quand un agent reçoit un message destiné à une instance dont l'identifiant est inconnu, une instance de la bonne classe de conversation est créée, la classe utilisée étant indiquée dans chaque message.

Les transitions des automates ne sont activées que par réception de message, sauf à l'initiation des conversations pour pouvoir envoyer le premier message. Les systèmes réalisés avec COOL sont donc uniquement réactifs.

2.2.3.3.2 AgenTalk

AgenTalk [Kuwabara *et al.*, 1995] poursuit le même but que COOL : fournir un langage de description et d'implémentation des protocoles de communicatoin entre les agents. Il s'en différencie en apportant un mécanisme d'héritage permettant de décrire de manière incrémentale ces protocoles.

AgenTalk est écrit en Common Lisp. Chaque protocole est défini d'un point de vue local à chaque agent par un automate étendu (automate simple étendu pour pouvoir utiliser des variables globales à l'automate dans les transitions) appelé un script. Les règles de transition sont des règles `SI condition ALORS action` qui permettent de décrire les automates. Les conditions possibles sont : « arrivée d'un message », « échéance d'un timeout » et « condition sur les variables de l'instance d'automate ». Pour implémenter un script, on dispose de deux macros Lisp : `define-script` pour déclarer le script, ses variables, son état initial et le script parent éventuel duquel hérite ce nouveau script ; et `define-script-state` pour décrire chaque état de l'automate. Le code exécuté lors du franchissement des transitions peut être personnalisé à l'aide de « fonctions d'agent » écrites par le programmeur de l'agent. Elles jouent le rôle de gestionnaires d'événements. Les fonctions d'agent constituent une fonctionnalité utile si l'on veut se constituer une bibliothèque de protocoles de communications personnalisables. Le script `mon_script` est un exemple de script. Il débute dans l'état `start`. Il y envoie à un autre agent, par l'intermédiaire d'une fonction d'agent, une tâche à sous-traiter puis passe dans l'état `wait_answer`. Lorsque la réponse `ok` arrive, le script se termine.

```
(define-script mon_script (task)
  :initial-state start
  :script-vars (ma_var))

(define-script-state (start mon_script)
  :on-entry
```

```
  (progn (setf ( ! send-task ($ task)))
        (goto-state wait_answer)))

(define-script-state (wait_answer mon_script)
  (:when (msg ok)
        :do (exit-script)))
```

Dans ce code, on peut également instancier d'autres automates. On peut ainsi réaliser une hiérarchie. La mémoire d'un automate parent est accessible depuis un fils. On peut partager de la mémoire entre un parent et un enfant mais également entre les enfants (puisqu'ils partagent l'accès à la mémoire de leur parent). Un agent peut être impliqué dans plusieurs protocoles simultanément, ceux-ci partageant les mêmes ressources. AgenTalk permet de décrire un mécanisme de résolution de conflits. L'héritage d'automate permet d'étendre un script déjà écrit pour lui ajouter d'autres états. Les auteurs avouent cependant eux-mêmes qu'utiliser l'héritage d'automate n'est pas chose aisée car il n'est possible que d'étendre un automate en ajoutant des états et qu'il est difficile de modifier par héritage les états existants.

2.2.3.4 Langages et architectures facilitant la gestion du temps dans les SMA

Nous venons de présenter les techniques de raisonnement sur le temps, les modèles d'agents permettant de combiner réactivité et cognition et les mécanismes de gestion de contextes de conversation. Nous proposons maintenant un tour d'horizon des autres langages et architectures d'agent présentant des fonctionalités de gestion du temps qui ont un intérêt pour notre étude.

2.2.3.4.1 Agent-0

Agent-0 est le langage proposé par Shoham en 1993 dans [Shoham, 1993] pour mettre en pratique ses idées sur la manière de concevoir et d'implémenter des agents. Cet article, souvent considéré comme celui ayant initié l'étude de la programmation orientée agent, propose d'utiliser une extension de la logique épistémique standard pour décrire l'état mental de l'agent et la théorie des actes de langage pour communiquer avec les autres agents.

Shoham pense que la programmation orientée agent doit être vue comme une extension de la programmation orientée objet et que l'« agenttitude » est dans la tête du programmeur. Nous pensons au contraire qu'il fait essayer de réifier le concept d'agent pour bénéficier des avantages du paradigme agent jusque dans la programmation.

Malgré cette différence de positionnement, le langage proposé par Shoham possède des caractéristiques intéressantes, notamment pour la gestion du temps. Il comporte des opérateurs pour les croyances, les obligations et les compétences des agents, chacun étant temporellement situé. Le temps est discrétisé. Du point de vue de l'agent, les actions ne s'inscrivent pas dans la durée. Du point de vue pratique, il faut qu'elles durent moins longtemps que le pas de temps, qui est déterminé une fois pour toute pour une application donnée. Il peut très bien être fixé à 50ms comme à plusieurs jours.

L'agent exécute les opérations suivantes à chaque top d'horloge :

1. lecture des messages en attente et mise à jour de l'état mental. Le programme de l'agent met ainsi à jour les croyances et les engagements de l'agent.
2. exécution des engagements pour la date courante. Cette étape est réalisée de manière automatique, sans que cela soit écrit dans le programme de l'agent.

La figure 2.5 décrit les flôts de contrôle et de données qui existent entre les différents modules d'un agent Agent-0.

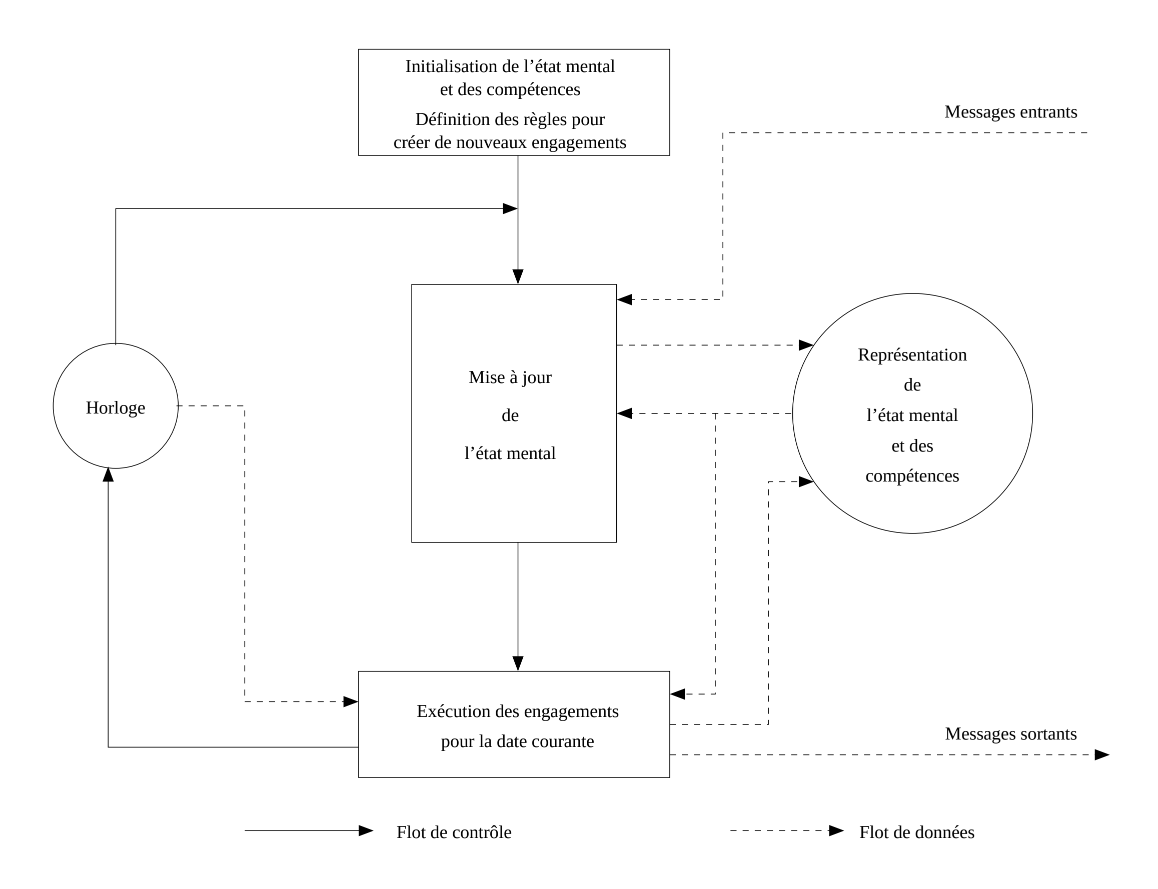


FIG. 2.5 – Diagramme de flot d'un agent Agent-0.

Agent-0 est présenté par Shoham comme un langage simple pour illustrer ses idées sur la programmation orientée agent. Il a cependant été surpris par certaines difficultés apparues à la conception et également par le nombre important d'exemples d'utilisation découverts. Nous détaillons l'exemple de réservation de billets d'avions auprès d'une compagnie aérienne proposé par Shoham. Cela nous permettra d'identifier les particularités du langage.

La syntaxe utilisée est proche du Lisp. On dispose d'opérateurs pour les croyances (`B`), pour l'obligation (`OBL`), pour le choix (`DEC`) et pour les compétences (`CAN`). Tous ces opérateurs sont temporalisés : par exemple, $\mathtt{CAN}_{\mathtt{a}}^{\mathtt{t}}\mathtt{action}$ représente le fait que l'agent `a` est capable de réaliser `action` au temps `t`.

On peut également utiliser des actions privées ou de communication qui sont placées dans l'agenda de l'agent et exécutées automatiquement au bon moment :

– `(DO t action)` permet de réaliser `action` au temps `t`.
– `(INFORM t a fact)` permet d'informer l'agent `a` du fait `fact` au temps `t`.
– `(REQUEST t a action)` permet de demander au temps `t` à l'agent `a` de réaliser `action`.

On peut également décrire des règles d'engagement sous cette forme :

```
(COMMIT msg-cond mental-cond (action)*)
```

où `msg-cond` est une condition sur les messages entrants, `mental-cond` une condition sur l'état mental de l'agent et `(action)*` une liste d'actions à réaliser si les conditions sont vérifiées.

Un programme Agent-0 est en définitive constitué de la liste des compétences de l'agent, de ses croyances initiales et de la liste de ses règles d'engagement.

Nous définissons maintenant 3 macros qui seront utilisées dans la suite de l'exemple :

```
(issue_bp pass flightnum time) =>
  (IF (AND (B ((- time h) (present pass)))
           (B (time (flight ?from ?to flightnum))))
      (DO time-h (physical_issue_bp pass flightnum time))).
```

La compagnie aérienne édite les billets `h` heures avant le départ en effectuant l'action basique `physical_issue_bp`.

```
(query_which t asker q) =>
  (REQUEST t askee (IF (B q) (INFORM (+ t 1) asker q))).
```

Cette macro permet d'être informé des faits qui valident `q`. Si `q` contient une variable quantifiée universellement, toutes les instances de réponse à la requête `q` seront retournées.

```
(query_whether t asker askee q) =>
  (REQUEST t askee (IF (B q) (INFORM (+ t 1) asker q)))
  (REQUEST t askee (IF (B (NOT q))
                   (INFORM (+ t 1) asker (NOT q)))).
```

Cette macro permet de demander confirmation ou infirmation d'un fait à un autre agent.

Nous définissons maintenant l'agent compagnie aérienne. Nous devons pour cela définir les bases de croyances, de compétences et de règles d'engagements initiales. La base de croyances contient les horaires des vols sous la forme `(time (flight from to number))` et le nombre de sièges disponibles pour chaque vol sous la forme `(time (remaining_seats time1 flight_number seats))`.

La base des compétences est définie ainsi :

```
((issue_bp ?a ?flight ?time) true)

((DO ?time (update_remaining_seats ?time1 ?flight_number
```

```
                                ?additional_seats))
 (B (?time (remaining_seats ?time1 ?flight_number
                            ?current_seats)))).
```

La compagnie aérienne est capable d'éditer des billets et de mettre à jour le nombre de sièges disponibles pour un vol donné.

Les règles d'engagement sont les suivantes :

```
(COMMIT (?pass REQUEST (IF (B,?p) (INFORM ?t ?pass ?p)))
         true
         ?pass
         (IF (B,?p) (INFORM ?t ?pass ?p)))

(COMMIT (?cust REQUEST (issue_bp ?pass ?flight ?time))
        (AND (B (?time (remaining_seats ?flight ?n)))
             (?n>0)
             (NOT ((CMT ?anyone)
                   (issue_bp ?pass ?anyflight ?time))))
        (myself (DO (+ now 1)
                    (update_remaining_seats ?time
                                            ?flight -1)))
        (?cust (issue_bp ?pass ?flight ?time)))
```

La première permet de répondre à une requête d'information sur les vols. La seconde permet de prendre en compte un message de réservation d'un billet s'il y a encore des places disponibles pour le vol demandé.

Le tableau suivant donne un exemple d'échange entre l'agent compagnie aérienne et un agent client. Le client récupère auprès de la compagnie aérienne les informations sur les vols de Lille à Nice pour le 10 avril. Il choisit le vol 42 de 8h30 et valide sa réservation. La compagnie aérienne se charge enfin d'éditer le billet une heure avant le départ.

Agent	**Action**
dinont	query_which 1march/1h00 dinont airline (10april/?!time (flight lille nice,?!num)))
airline	(INFORM 1march/2h00 dinont (10april/8h30 (flight lille nice #42)))
airline	(INFORM 1march/2h00 dinont (10april/10h00 (flight lille nice #54)))
dinont	(REQUEST 1march/3h00 airline (issue_bp dinont #42 10april/8h30))
airline	(DO 10april/7h30 (issue_bp dinont #42 10april/8h30))

Agent-0 possède un certain nombre d'inconvénients comme sa synchronisation à l'horloge, mais c'est un langage simple avec lequel les programmeurs ne peuvent pas être surpris par les interactions entre les différentes instructions.

2.2.3.4.2 April

April (Agent Process Interaction Language [McCabe et Clark, 1995]) est un langage de programmation pour construire des applications d'Intelligence Artificielle distribuée. April peut se voir comme un langage concurrent basé sur les objets, ces derniers étant des processus. Il permet de programmer des applications distribuées et capables de répondre à des événements temps réels. April est un langage symbolique orienté processus. Il permet de définir des processus et leur permet de communiquer par un mécanisme de passage de messages.

Les données sont fortement typées. Des fonctionnalités d'ordre supérieur sont incorporées. Elles permettent la structuration des programmes et permettent de passer les fonctions et procédures d'un processus à un autre pour qu'il puisse les exécuter. April s'appuie sur la programmation orientée acteur, en l'étendant. L'expansion de macros permet de construire des couches de langage au-dessus du niveau basique. Chaque processus possède sa file d'attente de messages entrants. Des modèles de message (*patterns*) sont utilisés pour filtrer les messages entrants acceptés à un moment donné. Ainsi, on peut retirer de la file d'attente certains messages et en laisser d'autres. Quand aucun message ne correspond aux modèles, le processus est suspendu. Il sera réactivé lorsqu'un nouveau message arrivera. Comme il est très difficile de préserver l'ordre des messages dans un SMA distribué, pouvoir accéder à toute la file des messages en attente paraît être important. Avec le système proposé par April, le programmeur n'a plus à se soucier de reconstituer l'ordre, cela est fait automatiquement car quand un message arrive en avance par rapport à un autre, son traitement peut être différé.

L'exemple suivant décrit un serveur qui exécute indéfiniment une tâche `T` lorsqu'il reçoit un message lui indiquant de la réaliser avec les paramètres `params` puis qui renvoie un message indiquant que l'action a été réalisée.

```
server([any]{} ?T)
{
  repeat{
         [do,any?params] -> {
         T(arg) ;
         Done >> replyto
         }
  } until quit
} ;
```

Le modèle de message utilisé est `[do,any?arg]`. `any` indique que la variable `params` peut être de n'importe quel type. Lorsqu'à un moment du programme, on s'attend à recevoir plusieurs types de messages, on utilise le | (ou) sur les *patterns*. L'exemple suivant indique comment réaliser une tâche si on nous le demande et supprimer de la file tous les autres messages en répondant qu'on ne les comprend pas.

```
{
    [do,any?arg] -> {T(arg) ; done >> replyto }
  | any?Other -> [unrecognized,Other] >> replyto
}
```

April possède quelques fonctionnalités permettant de gérer le temps : on peut activer un processus à une date donnée et on peut paramétrer la durée d'attente de nouveau message à l'aide de timeouts. Cependant, comme il n'est pas possible de prédire la durée des opérations entreprises, April n'est pas adapté aux applications où le temps est une dimension critique. Il est possible de définir des *procedure abstractions* et des *pattern abstractions*. Ces fonctionnalités permettent respectivement de construire dynamiquement une nouvelle procédure et un nouveau modèle de message. Ainsi, en se partageant des *pattern abstractions*, les agents peuvent apprendre à reconnaître de nouveaux types de messages, ce qui revient à extraire de ces messages les valeurs des variables. En partageant des *procedure abstractions*, un agent peut exécuter du code de manière transparente à l'intérieur du code d'un autre agent. Par exemple, considérons un agent serveur qui fournit une correspondance entre le nom d'un agent et ses compétences. Un autre agent peut lui envoyer ses propres critères de choix qui permettent de faire le tri entre les réponses possibles.

Chaque tâche/plan d'un agent est exécuté dans un processus séparé. Un processus particulier gère la mémoire de l'agent. Les autres tâches lui envoient des messages pour récupérer ou mettre à jour les données. Ne pas pouvoir accéder directement à la mémoire de l'agent et devoir procéder par envoi de messages est une limitation importante pour les performances.

2.2.3.4.3 PRS

PRS (Procedural Reasoning System, [Ingrand *et al.*, 1992]) est une architecture générique pour représenter et raisonner sur les actions dans un domaine dynamique.

Un agent PRS est constitué :
- d'une base de données des croyances actuelles de l'agent,
- d'un ensemble des buts courants,
- d'une bibliothèque de plans, nommés des procédures ou Knowledge Areas (KA), qui décrivent les séquences de tests et d'actions à réaliser pour atteindre un but ou pour réagir à des événements en provenance de l'environnement. les KA disposent d'une condition d'invocation qui spécifie dans quelles situations le KA est utile. La condition d'invocation est composée de deux parties. La partie déclenchement est une expression logique décrivant les événements qui doivent survenir pour que le KA soit invocable. Elle peut porter sur des changements dans la base des buts et/ou sur la base de croyances. La partie contextuelle est une expression logique spécifiant les conditions devant être vraies pour que le KA soit exécutable.
- d'une structure pour les intentions, composée d'un ensemble partiellement ordonné de plans.

Un interpréteur se charge de manière cyclique de :
- mettre à jour la base de croyances et des buts en fonction des événements extérieurs et des informations postées à la fin du cycle précédent,
- sélectionner dans les KA dont les conditions d'activation sont vérifiées ceux qui vont être placés dans la structure d'intention,
- choisir une tâche de la racine de la structure d'intentions et exécuter une étape de cette tâche. Cela déclenche une action primitive, la formation d'un nouveau sous-but ou de nouvelles croyances qui seront intégrées dans les bases de croyances et de buts au prochain cycle.

Chaque intention de la structure d'intentions représente une pile de KA invoqués. L'exécution d'un KA entraîne la formation de sous-buts qui eux-même invoquent d'autres KA, ce qui forme une pile de KA à la manière d'une pile d'appels de procédures des langages de programmation traditionnels. Lorsque le système traite plusieurs tâches, il gère ces piles dynamiquement en exécutant, suspendant et relançant ces procédures à la manière d'un système d'exploitation.

PRS ne permet pas de prendre en compte des actions primitives qui s'inscrivent dans la durée, mais il possède une particularité intéressante puisque le niveau méta est réifié : les KA peuvent permettre de raisonner sur d'autres KA.

2.2.3.4.4 TÆMS

TÆMS est l'aboutissement des travaux du Multi-Agent Systems Lab de l'université du Massachussets sur les choix des agents en présence de contraintes temporelles. C'est une architecture complète pour la gestion du contrôle local des agents et les raisonnements au niveau organisation du SMA.

TÆMS signifie Task Analysis, Environment Modeling and Simulation. Il propose un langage de description des activités des agents sous forme d'un arbre hiérarchique. Celui-ci permet de décrire les tâches de l'agent à plusieurs niveaux d'abstraction, avec des délais possibles à chaque niveau. Les feuilles de l'arbre représentent les méthodes exécutables.

La figure 2.6 donne un exemple d'arbre utilisé par TÆMS issu de [Horling, 1999]. Dans celui-ci, nous voyons comment un agent peut faire du café[3]. Les rectangles arrondis représentent des actions abstraites et les rectangles droits des actions élémentaires.

Les nœuds correspondants à des actions abstraites peuvent être annotés pour permettre à l'ordonnanceur de savoir comment évoluent chacun des critères en fonction des sous-tâches réalisées. Par exemple, les nœuds `Faire le café` et `Acquérir les ingrédients` sont annotés avec `q_min`, qui signifie que toutes les sous-tâches doivent être réalisées pour pouvoir accroître la qualité (sémantique du ET).

Une autre annotation permet de soulever une des caractéristiques essentielles de TÆMS : `q_exactly_one`. TÆMS sera principalement utilisé dans des cas où il existe différentes alternatives pour réaliser une tâche. C'est le cas ici avec l'eau qui peut être prise chaude ou froide. L'ordonnanceur pourra ainsi choisir entre les deux en fonction de l'état des ressources et des critères fournis pour la recherche de solution.

Nous voyons également que certaines actions doivent être ordonnées grâce à la relation `Permet` : on ne peut pas moudre les grains de café avant de les posséder. Notons enfin que TÆMS permet de gérer des ressources qui sont utilisées par les différentes tâches. Les ressources peuvent être consommables ou non consommables. Il est possible d'indiquer les limites hautes et basses ainsi que le niveau

[3] Ce qui a actuellement une raisonnance particulière pour l'auteur.

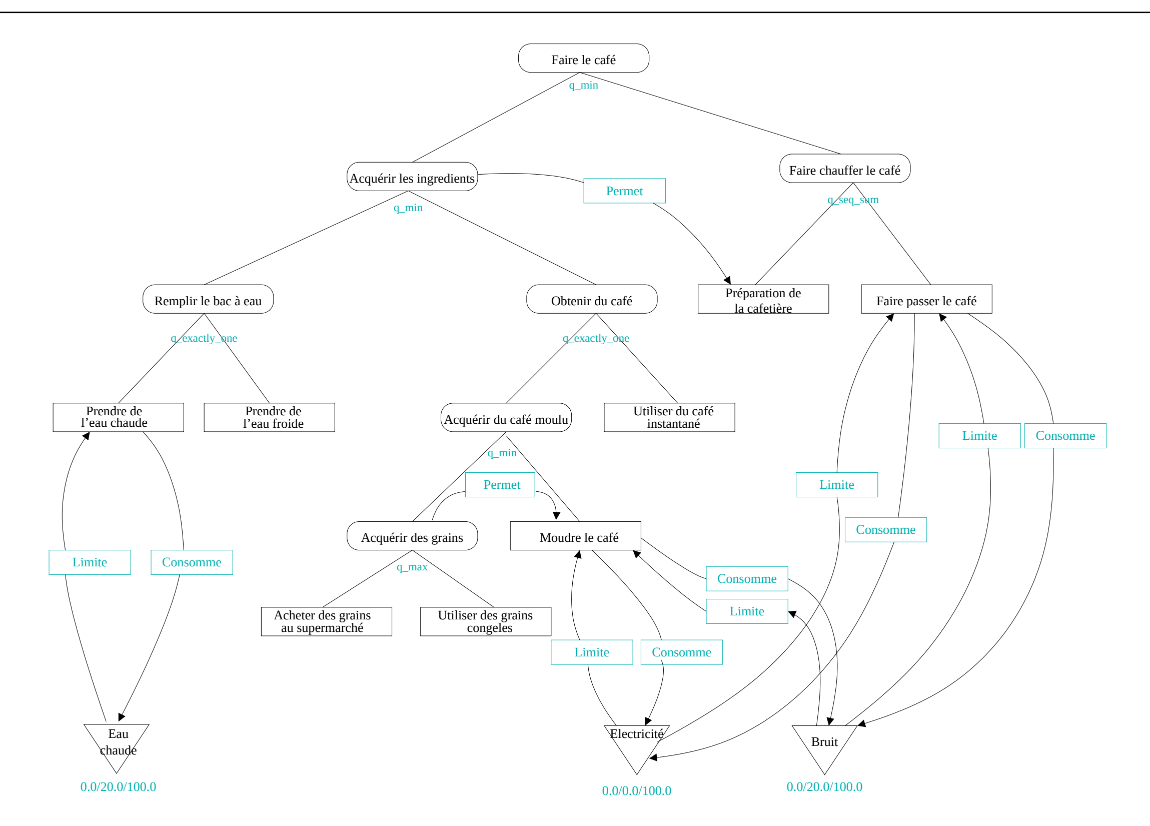


FIG. 2.6 – Exemple d'arbre TÆMS.

courant pour chaque ressource. Ici, l'eau chaude est actuellement au niveau 20. Le niveau minimal est 0 et le niveau maximal est 100. Les annotations et relations possibles sur les nœuds sont très nombreuses et permettent de modéliser des situations très complexes.

L'ordonnanceur utilisé est Design-to-Time ou son évolution Design-to-Criteria. Le principe de l'ordonnancement Design-to-Time est de construire dynamiquement des plans d'exécution en privilégiant les plans qui, par ordre de priorité, aboutissent à des qualités nulles pour tous les groupes de tâches, maximisent la somme des qualités de tous les groupes de tâches, et minimisent le temps total d'exécution des méthodes. Le résultat de cet algorithme d'ordonnancement est un plan d'exécution qui spécifie quelles méthodes exécuter, quand les exécuter et quelles valeurs sont attendues après leur exécution.

Design-to-Time a été utilisé pour ordonnancer des algorithmes anytime [Garvey et Lesser, 1996] en choisissant quelques temps d'exécution et la qualité correspondante dans le profil de performance afin de les considérer comme des méthodes différentes pour réaliser une tâche. Cette même étude a mis en évidence la possibilité de considérer Design-to-Time comme un algorithme anytime. Design-to-Time utilise des heuristiques pour réduire la complexité initiale de $O(n!)$ à une complexité polynomiale.

Design-to-Criteria peut être vu comme une généralisation de Design-to-Time. Le temps n'est plus le seul critère qui guide la recherche de solutions. L'agent dispose d'une fonction objectif à atteindre.

Elle peut être exprimée comme des contraintes sur la qualité, le coût et la durée de chaque tâche. On fournit également un ensemble de paramètres qui permettent à Design-to-Criteria de diriger sa recherche dans la bonne direction. Le problème a une complexité très importante et l'algorithme a été très fortement optimisé. Il est capable de trouver une solution à des problèmes ayant une centaines de tâches en quelques secondes. [Vincent *et al.*, 2001] utilise Design-to-Criteria dans un cadre temps réel mou.

TÆMS a été intégré dans un framework pour la coordination entre agents nommé MQ. Dans ce système, la coordination est centrée ordonnancement et utilise GPGP (Generalized Partial Global Planning [Decker, 1995]). La figure 2.7 donne une vue simplifiée de l'architecture d'un agent utilisant GPGP, MQ et TÆMS. GPGP utilise une base de croyance, un ordonnanceur local et un module de coordination pour la communication. Chaque agent construit sa propre vision partielle de la sous-tâche qu'il doit accomplir. L'ordonnanceur prend aussi bien en compte les effets locaux que non-locaux des actions. La partie MQ quant à elle modélise et raisonne sur les tâches au niveau organisationnel.

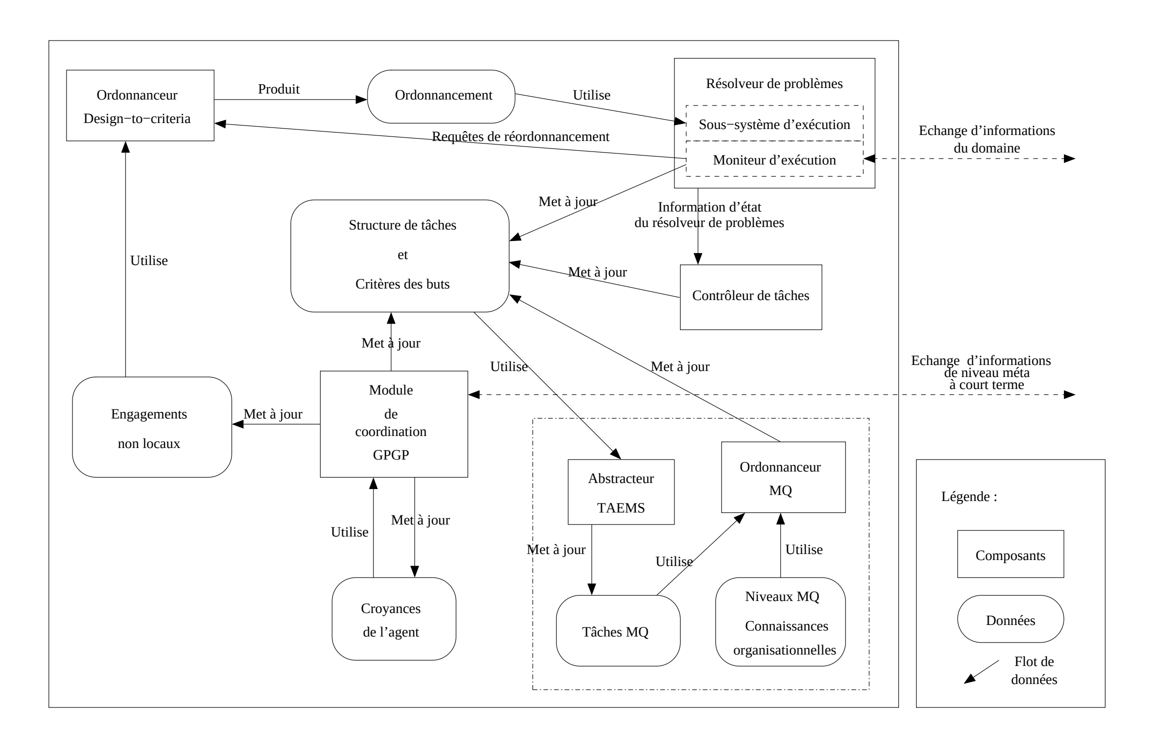


FIG. 2.7 – Architecture d'un agent TÆMS.

TÆMS dispose d'une documentation très complète, en commençant par les articles sur Design-to-Time [Garvey et Lesser, 1996] et Design-to-Criteria [Wagner et Lesser, 2000]. Une description et une analyse très complète pourra être trouvée dans la très volumineuse thèse de Thomas Wagner [Wagner, 2000]. On y trouvera en particulier une explication sur la structure de boucle, qu'il n'est pas facile à introduire dans TÆMS.

TÆMS présente quelques inconvénients qui en limitent l'usage dans notre cadre. En effet, Design-to-Time, Design-to-Criteria et la structure de tâches ne sont intéressants que quand on dispose de plu-

sieurs alternatives pour atteindre un but. L'impossibilité de décrire facilement des traitements itératifs ou récursifs est également une limitation importante. Enfin, TÆMS ne propose aucun outil pour réduire le fossé entre le niveau agent représenté par la structure de tâches et le niveau du code hérité contrôlé par le niveau agent.

2.2.3.4.5 Temporal Agent Programs

Temporal Agent Program (TAP) [Dix *et al.*, 2001] est une extension de la plate-forme IMPACT [Arisha *et al.*, 1999, Subrahmanian *et al.*, 2000, Eiter et Subrahmanian, 1999]. IMPACT permet de construire des agents au-dessus de tout type de code hérité.

Pour cela, on décrit tout d'abord la base de code sur laquelle vont être basés les agents : les types de données et les fonctions fournies. L'état d'un agent à un instant `t` est ensuite décrit par l'ensemble des objets de la base de code référencées dans les structures de données de l'agent.

Un agent est décrit par :
- un ensemble de contraintes d'intégrité que l'état d'un agent doit vérifier pour pouvoir être considéré comme valide,
- une notion de concurrence qui spécifie comment on peut combiner un ensemble d'actions en une seule action,
- un ensemble de contraintes d'action qui définit les circonstances dans lesquelles des actions peuvent s'exécuter de manière concurrente,
- un programme d'agent qui détermine les actions que l'agent doit, peut ou ne peut pas entreprendre. On utilise pour cela les opérateurs déontiques `P`, `F`, `O`, `W` et `Do`, respectivement pour la permission, l'interdiction, l'obligation, l'annulation d'une obligation et la réalisation actuelle d'une action.

Les premières versions d'IMPACT possèdent quelques limitations : les actions n'ont pas de durées et on est incapable de les placer dans un agenda pour qu'elles soient exécutées dans le futur. Temporal Agent Program lève ces limitations.

Dans TAP, les actions s'inscrivent donc dans la durée. L'état associé à une action évolue de manière continue. C'est le cas par exemple de la position d'un véhicule qui réalise l'action `drive(Lille, Paris)`.

On peut décrire des instants intermédiaires, des *checkpoints*, auxquels on peut mettre à jour l'état actuel de l'action de manière incrémentale. Les *checkpoints* n'interrompent pas les actions. Ils peuvent être rapprochés des interruptions utilisées dans les processeurs et les systèmes d'exploitation, en ce sens qu'ils obligent certaines actions à être entreprises quand le *checkpoint* arrive. Une *checkpoint expression* décrit les *checkpoints* associés à une action. Ils peuvent être spécifiés de manière absolue ou relative par rapport à la date de début de l'action. On peut également les associer à des événements comme la réception d'un message.

La mise à jour d'un état est spécifiée par un *timed effect triple* défini ainsi : <cpe, Add, Del>, où cpe est une *checkpoint expression*, et où Add et Del spécifient les modifications à effectuer sur l'état sous forme de listes de faits à ajouter ou supprimer.

On décrit une action qui s'inscrit dans la durée avec :
- un nom sous la forme α(X1, ..., Xn),
- un schéma sous la forme (T1, ..., Tn) qui indique que la variable Xi doit être du type Ti,
- une condition d'appel de code, appelée la précondition de l'action,
- une expression qui définit la durée de l'action,
- un ensemble de *timed effect triple*.

L'historique des états est conservé et il est possible de l'utiliser dans le programme de l'agent, tout comme l'historique des actions entreprises par l'agent. Il est possible de spécifier dans le programme de l'agent des actions a réaliser dans le futur. Cela se fait en ajoutant des annotations temporelles aux préconditions des actions. Celles-ci peuvent être spécifiées sous forme de dates ou d'intervalles entre deux dates.

TAP possède bon nombre de propriétés intéressantes dans notre cadre : on dispose d'une encapsulation de code hérité qui permet de raisonner dessus, les actions s'inscrivent dans la durée, on peut suivre leur déroulement et prendre des décisions en fonction des informations conservées dans l'historique, ... Il est cependant dommage que ce système soit complètement décorellé du système d'exploitation.

2.2.3.4.6 DECAF

DECAF (Distributed, Environment-Centered Agent Framework) [Graham *et al.*, 2003, Graham, 2001] est une infrastructure pour le développement de systèmes multi-agents développée à l'université du Delaware.

DECAF propose de programmer les agents de manière graphique dans un module nommé Plan-Editor. Celui-ci permet de décrire un réseau de tâche hiérarchique semblable aux structures de tâches de TÆMS et l'exécution des plans est dirigée à la manière de RETSINA (Reusable Environment for Task-Structured Intelligent Networked Agents) [Williamson *et al.*, 1996].

DECAF peut être vue comme une infrastructure agent de seconde génération. La première génération s'est plutôt focalisée sur les API de gestion du cycle de vie des agents et sur la communication inter-agents. La seconde génération apporte une vision plus haut niveau en proposant une architecture complète pouvant être vue comme un « système d'exploitation » pour systèmes multi-agents.

L'architecture des agents ainsi que l'architecture interne de DECAF sont très modularisés et chaque module s'exécute dans un processus léger distinct. Cela autorise par exemple de pouvoir très facilement déployer un simple agent sur une plateforme multi-processeurs. Cependant, comme nous

l'avons signalé plus haut dans la section 2.2.2.1.1, cela complique énormément l'écriture du code des agents.

DECAF est intéressant lorsque l'on veut rapidement développer des systèmes multi-agents de manière visuelle mais il ne propose pas d'outils pour contrôler de manière fine le lien entre le niveau agent et le niveau système.

2.3 Synthèse

Nous avons décrit le problème que nous voulons résoudre dans cette thèse : nous voulons intégrer tous types d'algorithmes complexes dans nos SMA, en particulier ceux issus du domaine de l'Intelligence Artificielle. Nous voulons que cela se fasse de manière à respecter des propriétés que nous attribuons à nos agents : autonomie, extraversion et conscience du temps. Enfin, nos agents doivent être capables de se partager les ressources processeur disponibles, qu'elles soient centralisées ou distribuées.

Nous avons ensuite rappelé ce qu'est un algorithme et quelles sont les classifications qui ont été proposées. Nous avons décrit comment il est possible de résoudre des problèmes ayant des complexités élevées en des temps raisonnables et contrôlables en utilisant des heuristiques et des algorithmes incrémentaux, voire même anytime.

Nous avons parcouru les différentes techniques proposées pour distribuer des calculs dans plusieurs fils d'exécution, ceux-ci s'exécutant sur une unique machine ou dans un environnement distribué. Nous avons remarqué que la programmation de systèmes distribués est très complexe et qu'il convient d'utiliser des outils qui permettent de découpler au maximum les différentes entités et qui suppriment au maximum l'indéterminisme.

Nous nous sommes penchés sur les méthodes de gestion et de raisonnement sur le temps dans les systèmes intelligents. Les techniques de base permettent de raisonner sur le temps de manière discrète ou continue et même sur des durées incertaines, mais elles ne prennent pas en compte la dualité cognitif/réactif nécessaire au bon fonctionnement d'un SMA. Cette dualité est prise en compte par les modèles d'agent hybrides.

Enfin, nous avons identifié un certain nombre de fonctionnalités intéressantes pour notre étude dans les langages et architectures d'agents proposés dans la littérature. C'est le cas par exemple de la gestion très simple de l'agenda dans Agent-0, de la structure de tâche de TÆMS annotée de telle manière que l'agent peut raisonner sur les différentes alternatives en fonction de ses contraintes, ou encore de la manière avec laquelle Temporal Agent Programs lie un langage logique à du code hérité.

Chapitre 3

Des outils de calcul pour les agents

Un mauvais ouvrier a toujours de mauvais outils.
Proverbe.

Si l'homme ne façonne pas ses outils, les outils le façonneront.
Arthur Miller.

Mains, outils de l'esprit sans lesquels la pensée n'est que chimère.
Alain Aslan.

Dans ce chapitre, nous nous demandons comment doivent être considérées les fonctionnalités de calcul des agents. Nous ne nous préoccupons pas des problèmes de gestion des délais donnés aux tâches de calcul, qui seront traités dans le prochain chapitre. Nous nous intéressons plus particulièrement à la manière de les encapsuler pour faciliter leur manipulation par les agents. Dans ce cadre, nous aimerions pouvoir fournir des *modes d'emploi* des algorithmes utilisés sur lesquels les agents pourront raisonner. Les algorithmes ne seront ainsi plus considérés comme des boîtes pour lesquelles on ne dispose d'aucune information sur le déroulement du calcul (des boîtes noires). Ils seront au contraire considérés comme des boîtes pour lesquelles on dispose de certaines informations sur le déroulement du calcul (des boîtes translucides). Nous voulons également trouver un moyen pour que les agents puissent lancer de multiples tâches de calcul tout en conservant un comportement extraverti, c'est-à-dire qu'il doivent rester capables de prendre en compte les changements de l'environnement. Concernant ce dernier point, nous nous attacherons à ce que la solution permette une programmation aisée des agents.

Les travaux de cette partie se basent sur la notion d'artifact introduite dans [Viroli et Ricci, 2004] et décrite dans la section 3.2. Nous réutilisons le formalisme initialement introduit par Omicini, Ricci et Viroli. Il est décrit dans la section 3.4. Je propose dans la section 3.3 une nouvelle classe d'artifacts

pour l'encapsulation des calculs longs : les artifacts de calcul. Je présente dans la section 3.4.3 les extensions que nous avons apportées au formalisme pour le flexibiliser et le rendre compatible avec les artifacts de calcul. Je propose dans la section 3.4.4 une classification des algorithmes que nous encapsulons dans les artifacts de calcul et je fournis les « modes d'emploi » qui permettent de les contrôler.

3.1 Position du problème

3.1.1 Besoin d'un nouveau type d'entité

Nous voulons que les agents conservent un comportement extraverti[4] tout en étant capable d'exécuter des tâches calculatoires longues. De plus, pour les raisons citées dans le chapitre précédent, nous ne pouvons modifier le code qui réalise les tâches calculatoires. En particulier, nous ne pouvons pas intégrer dans ce code un appel régulier à la méthode de traitement des nouveaux messages de la boîte aux lettres de l'agent. Il faut donc disposer de plusieurs fils d'exécution qui s'exécutent en parallèle (processus ou processus léger) :

- un qui intègre le comportement extraverti de l'agent (traitement des messages, vérification des changements dans l'environnement),
- un autre pour chaque tâche de calcul longue.

La question à laquelle il faut répondre maintenant est : sous quelle forme doit-on faire apparaître le second type de fil d'exécution dans le SMA ? Faut-il le faire apparaître comme une partie des agents : un objet, un composant, une couche ? Ou comme un type d'agent particulier ? Ou encore comme une nouvelle entité dans les SMA ?

3.1.2 Une partie de l'agent

Considérons tout d'abord que les fils d'exécution qui réalisent les tâches longues soient des parties de l'agent.

On pourrait modéliser un agent comme un assemblage d'objets ou de composants. L'approche par composant semble séduisante car elle permet l'utilisation de code hérité et elle apporte la réutilisabilité dans différents contextes. De plus, les communautés composant et agent tentent de combiner leurs méthodes pour allier les avantages des deux approches : autonomie des agents et maturité des composants (par ex. : [Lacouture et Aniorté, 2006]). Cependant, les spécifications qui accompagnent un composant portent principalement sur les noms et paramètres des appels de méthode, sans donner un mode d'emploi qui indiquerait à l'agent quel enchaînement utiliser pour atteindre ses buts. Ces spécifications très bas niveau ne permettent pas non plus de déduire les propriétés temporelles des fonctionnalités du composant.

[4] Voir section 2.1.1 pour la définition précise que nous donnons à ce terme.

Le problème de la réactivité, au sens de la prise en compte *à temps* des changements dans l'environnement, tout en ayant des raisonnements à long terme, a déjà été traité. Nous avons notamment présenté dans la section 2.2.3.2 les principaux modèles à couches proposés : InTeRRaP et Touring Machines. Nous rappelons que ces modèles présentent chacun trois couches qui interagissent entre elles pour décider de la prochaine action à réaliser. D'autres modèles ont été également proposés pour combiner des capacités cognitives et réactives comme [Guessoum et Dojat, 1996] ou [Bussman et Demazeau, 1994]. Tous ces modèles permettent de trouver un compromis entre les réactions de type réflexe et les réactions délibératives, mais ils ne sont pas adaptés lorsque les actions que l'agent réalise sont elles-mêmes des délibérations longues sur lesquelles les agents doivent raisonner, ce qui est notre cas.

Les modèles proposés (à composants ou à couches) ne conviennent pas à nos besoins. On peut donc se demander si nous devons proposer un nouveau modèle d'agent (à couche ou autre) qui réponde à nos attentes. En réalité, les problèmes que nous voulons résoudre (encapsulation des tâches de calcul et conservation de l'extraversion des agents) sont suffisamment généraux pour que l'on veuille proposer une solution indépendante du modèle d'agent choisi. Il faudrait donc plutôt utiliser d'autres entités du SMA pour exécuter les tâches calculatoires.

3.1.3 Un second agent

S'il vaut mieux que les délibérations longues soient exécutées par une autre entité, considérons qu'elles le sont par un autre agent.

C'est ce que nous avons proposé dans [Dinont *et al.*, 2006b]. Nous avons défini deux classes d'agents : une au comportement extraverti qui ne peut s'engager dans des traitements longs et une seconde au comportement introverti qui ne se remet en phase avec son environnement qu'après avoir terminé l'exécution de ses tâches. Les agents extravertis doivent déléguer leurs tâches longues aux agents introvertis. De plus, pour gérer des dates limites sur les tâches, l'exécution des agents introvertis est commandée par les agents extravertis. Un agent extraverti négocie des tranches de temps pour des tâches calculatoires réalisées par des agents introvertis auprès d'un agent de services temporels (AST). L'utilisation de l'AST permet de garantir que les tâches se terminent avant leur date limite. Les agents extravertis ont la charge de démarrer et de mettre en pause les agents introvertis qui travaillent pour eux.

Cette solution, correcte, pose cependant plusieurs problèmes conceptuels dûs principalement au fait que l'on attribue généralement aux agents la propriété d'autonomie.

La question de l'autonomie a déjà été beaucoup traitée et de nombreuses définitions ont été proposées [Luck et d'Inverno, 2001, Carabelea *et al.*, 2003, Joseph et Kawamura, 2001]. Nous posons ici une fois de plus ce problème dans le but d'aboutir à une vision pragmatique de l'autonomie qui peut avoir un sens du point de vue opérationnel et qui s'adapte bien à notre cadre d'étude.

Le problème provient du fait que l'on présente le paradigme agent comme une solution pour casser la complexité structurelle des systèmes, et ceci grâce à la propriété d'autonomie des agents. L'autonomie serait ainsi le gage d'un couplage le plus faible possible entre les agents[5].

Mettons nous maintenant à la place d'un programmeur fraîchement initié au paradigme agent. Il existe un trop grand fossé entre le concept général d'autonomie et le code qu'il peut écrire pour ses agents. Il en reviendra à se poser le même type de questions que quand nous parlions dans la section 1.1 du sens à donner à l'intelligence lorsque celle-ci est implantée sous forme de programme dans un ordinateur : comment programmer un agent pour qu'il soit autonome ? Comment vérifier que l'agent que j'ai écrit peut être considéré comme autonome ?

Nous voulons que notre discours autour de l'autonomie permette facilement de trouver des réponses aux questions précédentes. Le concept d'autonomie étant très vaste et notre problématique étant de type génie logiciel, nous avons identifié une conséquence opérationnelle de l'autonomie qui nous permet de manipuler ce concept abstrait de manière concrète dans nos applications :

Un agent A est considéré comme étant autonome par rapport à un agent B si l'agent B ne peut prédire à coup sûr le comportement de l'agent A.

Avant de continuer, il convient de préciser le sens que nous donnons ici à un autre terme : comportement. Du point de vue d'un agent B, le comportement d'un agent A sera défini par ce que perçoit B de A. Comme nos agents n'interagissent que par envoi de messages, prédire le comportement de A revient à prédire quand et quels messages seront transmis de A à B. Le comportement est en grande partie composé des *réactions* de A aux requêtes de B. L'expression de l'autonomie de A dans son comportement apparaît par des réactions imprévues. Par exemple, A peut ne pas répondre à un message de B ou y répondre en dépassant les délais.

Nous pouvons déduire de cette conséquence opérationnelle de l'autonomie qu'un agent ne dispose donc que d'un modèle approximatif d'un autre agent. S'il disposait d'un modèle précis, l'autre agent ne pourrait plus être considéré comme autonome par rapport à lui. Or, nous avons décrit précédemment nos tâches calculatoires comme des boîtes translucides : un algorithme vu comme une boîte noire non modifiable accompagné de ses caractéristiques et de son mode d'emploi. Un algorithme n'est pas vu ici comme une entité autonome et ses caractéristiques sont un modèle précis sur lequel un agent peut compter pour effectuer ses raisonnements.

Lorsque l'on programme un agent, il faut prendre en compte l'autonomie des autres agents. Dans notre cas, cette idée s'exprime par le fait que l'on devrait se refuser à arrêter et redémarrer un autre agent, celui-ci perdant son autonomie par rapport à l'agent qui le commande. Or, nos agents introvertis sont commandés par les agents extravertis. D'un point de vue conceptuel, le fil d'exécution qui exécute nos tâches calculatoires se détache donc de l'idée que l'on se fait du concept d'agent.

[5]Il est important de noter que si l'on met autant en avant le concept d'autonomie, la complexité qu'on essaie de supprimer de la vision globale du système peut se reporter dans la conception de chacun des agents ou dans les interactions entre les agents. Il est dès lors important de trouver les limites à ne pas dépasser et trouver un modèle d'agent qui favorise l'autonomie tout en restant simple.

Nous pensons que l'autonomie est le point d'orgue du paradigme agent qui lui permet de se différencier du paradigme objet au niveau de l'implémentation des systèmes. Si on ne prend pas garde de préserver l'autonomie jusqu'au bout, on n'aura eu les bénéfices de la programmation orientée agent que dans la phase de modélisation et une bonne partie de ses bienfaits auront disparus dans l'implantation finale.

La solution utilisant deux classes d'agents ne paraît pas satisfaisante et nous devons nous résoudre à voir si l'on aurait un intérêt à introduire un nouveau type d'entités dans les SMA.

3.1.4 Une nouvelle entité ?

L'introduction d'un nouveau type d'entités dans les SMA est délicate. Il faut identifier de manière précise un nouveau concept suffisamment différent de celui d'agent, ce qui est difficile tant le concept d'agent est abstrait. Il est d'ailleurs très souvent possible de s'en sortir en identifiant plusieurs classes d'agents différentes, comme ce qui a été proposé dans le paragraphe précédent. Ferber avait notamment proposé dans [Ferber, 1995] une classification des agents en fonction de leur type de comportement qui est pertinente dans notre cas. Elle est rappelée dans la figure 3.1.

Relation au monde / Conduites	Agents cognitifs	Agents réactifs
Téléonomiques	Agents intentionnels	Agents pulsionnels
Réflexes	Agents « modules »	Agents tropiques

FIG. 3.1 – Les différents types d'agents définis par Ferber

Pour Ferber, ce qui différencie les agents cognitifs des agents réactifs, c'est leur relation au monde. Un agent cognitif dispose d'une représentation symbolique du monde à partir de laquelle il peut raisonner. Un agent réactif ne dispose que d'une représentation sub-symbolique du monde, c'est-à-dire limitée à ses perceptions. Nous ne nous intéressons qu'à la colonne correspondant aux agents cognitifs. Les agents extravertis décrits dans le paragraphe précédent sont des agents intentionnels car ils ont un comportement téléonomique (leur comportement est dirigé par des buts *explicites*). Au contraire, les agents introvertis précédemment cités entrent plutôt dans la catégorie des agents « modules » : ils sont utilisés par les agents extravertis comme des sous-traitants dociles qui exécutent de manière réflexe les travaux qu'on leur donne. Ferber concède que ces agents se situent à la limite du concept d'agent. Nous pouvons nous poser la question de savoir ce que sont des agents « modules » qui ne seraient pas autonomes, comme nos boîtes translucides. Faut-il toujours les considérer comme des agents ou y a-t-il un bénéfice à les considérer comme un nouveau type d'entité ? Les remarques du paragraphe précédent sur le manque d'autonomie, ainsi que le fait que l'on doive disposer d'un modèle précis des entités qui réalisent les tâches calculatoires, sont des arguments suffisamment forts pour se permettre d'introduire un nouveau type d'entité dans les SMA. Celle-ci ne sera pas autonome, pourra être utilisée par les agents comme outil de calcul pour atteindre leurs buts et disposera d'un mode d'emploi sur lequel l'agent pourra compter et raisonner.

3.2 Les artifacts

Il est apparu pendant l'étude que les travaux du groupe de recherche aliCE basé au laboratoire DEIS de l'université de Bologne, portant sur une problématique différente, s'adaptent parfaitement à notre situation.

Leurs travaux portaient au départ sur la coordination entre agents. Ils se sont aperçus qu'il y avait un gain du point de vue conceptuel et du point de vue génie logiciel à externaliser toute la mécanique qui sert à la coordination entre les agents. Ils ont ainsi introduit la notion d'*artifact de coordination* [Omicini *et al.*, 2004] et l'ont placé comme un autre type d'entité de premier plan à côté des agents. Ils ont ensuite cherché à étendre le concept à d'autres utilisations que la coordination et à identifier les modifications que l'utilisation des artifacts apportent à la modélisation et à la programmation des systèmes multi-agents [Ricci *et al.*, 2005, Ricci *et al.*, 2006].

Nous décrivons dans la suite le concept d'artifact[6] tel que nous l'utilisons dans nos systèmes. Il y a quelques différences par rapport à la définition initialement introduite par l'équipe italienne dans le cadre de ses travaux sur la coordination. Celles-ci sont mises en évidence dans le texte. Nous proposons ensuite une spécialisation du concept pour les longs calculs. L'analyse du langage utilisé pour spécifier les modes d'emploi des artifacts fait l'objet d'une section particulière.

3.2.1 Définition

Omicini remarque dans [Omicini *et al.*, 2005] que l'utilisation d'outils a toujours accompagné l'évolution de l'espèce humaine, depuis l'homo-habilis à l'homo-sapiens-sapiens, en passant par l'homo-faber, l'homme « fabricateur ». Il est même certain aujourd'hui que le développement de l'intelligence comme caractéristique de l'espèce humaine est très fortement lié à la disponibilité et au développement des outils. La théorie de l'activité humaine [Kaptelinin *et al.*, 1995] fait ressortir le fait que les outils permettent de structurer l'environnement, qu'ils sont utilisés dans toute activité humaine et qu'il est même possible de comprendre les activités humaines simplement en observant les outils de médiation utilisés. Si l'on transpose cela aux agents et à leur environnement, nous pouvons dès lors faire l'hypothèse que placer la notion d'outil comme type d'entité de premier plan au sein des systèmes multi-agents ne peut être que bénéfique. Cela nous permettra tout du moins de mieux structurer les systèmes et d'identifier les améliorations possibles au niveau des outils que les agents utilisent.

Si un agent est généralement vu comme un système dirigé ou orienté par des buts, un artifact est une entité qu'un agent *utilise* pour atteindre ses buts. Un artifact persiste au sein du système indépendamment du cycle de vie des agent. De manière plus précise, un artifact est caractérisé par :

– *une interface d'usage :* définie comme un ensemble d'opérations. Les opérations sont de deux types : l'exécution d'une action et la perception de la terminaison d'une action.

[6] Nous conservons ici l'orthographe « artifact » pour éviter la confusion avec l'acception usuelle du mot artefact de la langue française : phénomène d'origine humaine.

– *sa fonction :* la description du ou des services rendus par l'artifact. Elle peut être utilisée pour rechercher et découvrir dynamiquement des artifacts.
– *un ensemble d'attributs :* ce sont des variables internes que l'agent exhibe à l'extérieur.
– *un ensemble d'instructions opératoires :* la description (formelle) de la manière dont les agents peuvent utiliser l'artifact. Cela constitue le mode d'emploi de l'artifact.
– *une spécification de son comportement :* la description (formelle) du comportement interne de l'artifact non perceptible de l'extérieur.

La figure 3.2 montre comment on représente un artifact et ses caractéristiques associées.

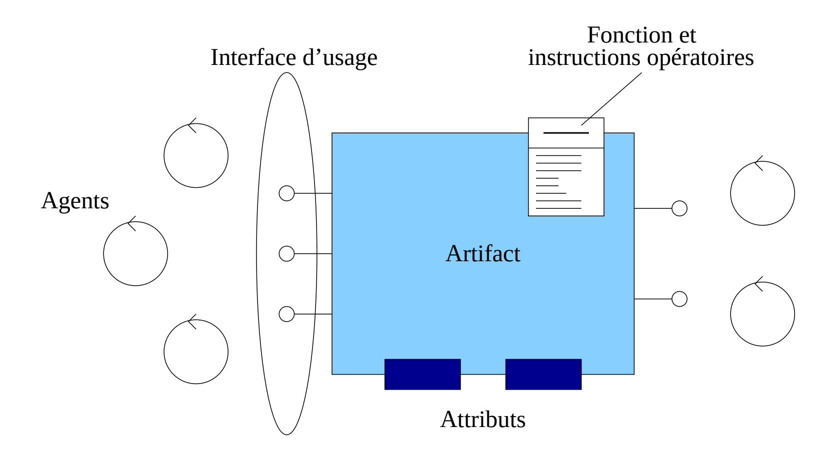


FIG. 3.2 – Un artifact.

L'interface d'usage indique la nature des échanges entre les agents et les artifacts. Ceux-ci sont dissymétriques puisque les agents peuvent exécuter des actions et les artifacts peuvent fournir des perceptions de la terminaison d'une action. Un agent qui décide d'exécuter une action *oblige* l'agent à l'exécuter *immédiatement*. C'est un lien quasi-physique comme quand on appuie sur un bouton pour démarrer un appareil électrique. Les communications sont ainsi d'un type totalement différent de la communication entre agents, qui se fait par actes de langage. On retrouve d'ailleurs cette distinction dans la programmation orientée objet : lorsqu'on appelle une méthode d'un autre objet, celui-ci est obligé de l'exécuter immédiatement alors que lorsqu'on utilise la programmation événementielle, les messages peuvent être considérés de manière asynchrone ou même être complètement ignorés.

Les attributs n'existent pas dans la définition de base proposée par l'équipe italienne. Nous les ajoutons pour fournir de manière simple la propriété d'**inspectabilité** indiquée dans [Omicini *et al.*, 2004]. Pour être inspectable, un artifact doit exhiber des propriétés qui peuvent être statiques ou dynamiques. Les propriétés statiques comprennent l'interface d'usage, les instructions opératoires et la fonction. Les propriétés dynamiques sont représentées par les attributs. Ceux-ci ont deux fonctions. Ils permettent tout d'abord aux agents de configurer un artifact avant de l'utiliser. Ils permettent ensuite l'inspectabilité des propriétés dynamiques de l'artifact pendant son utilisation. A ce moment, ils passent en lecture seule pour éviter les conflits d'accès lorsque l'artifact veut mo-

difier les valeurs de ses attributs. Nous considérons que l'état actuel des instructions opératoires de l'artifact constitue un attribut obligatoire.

La spécification du comportement peut permettre de faire des vérifications formelles sur le fonctionnement interne de l'agent. Nous n'utilisons pas cette possibilité dans nos systèmes.

A nos yeux, la caractéristique la plus importante d'un artifact réside dans ses instructions opératoires. Celles-ci permettent de définir de manière formelle le mode d'emploi de l'artifact. Ainsi, le comportement de l'artifact est **prévisible**. L'artifact s'engage implicitement à respecter son mode d'emploi. Nous consacrons la section 3.4 à l'étude des instructions opératoires.

3.2.2 Utilisations possibles

La figure 3.3 illustre un certain nombre d'utilisations possibles des artifacts. Remarquons qu'un artifact peut être considéré selon différents angles et donc appartenir à plusieurs catégories. Cette classification n'est donc pas une taxonomie[7].

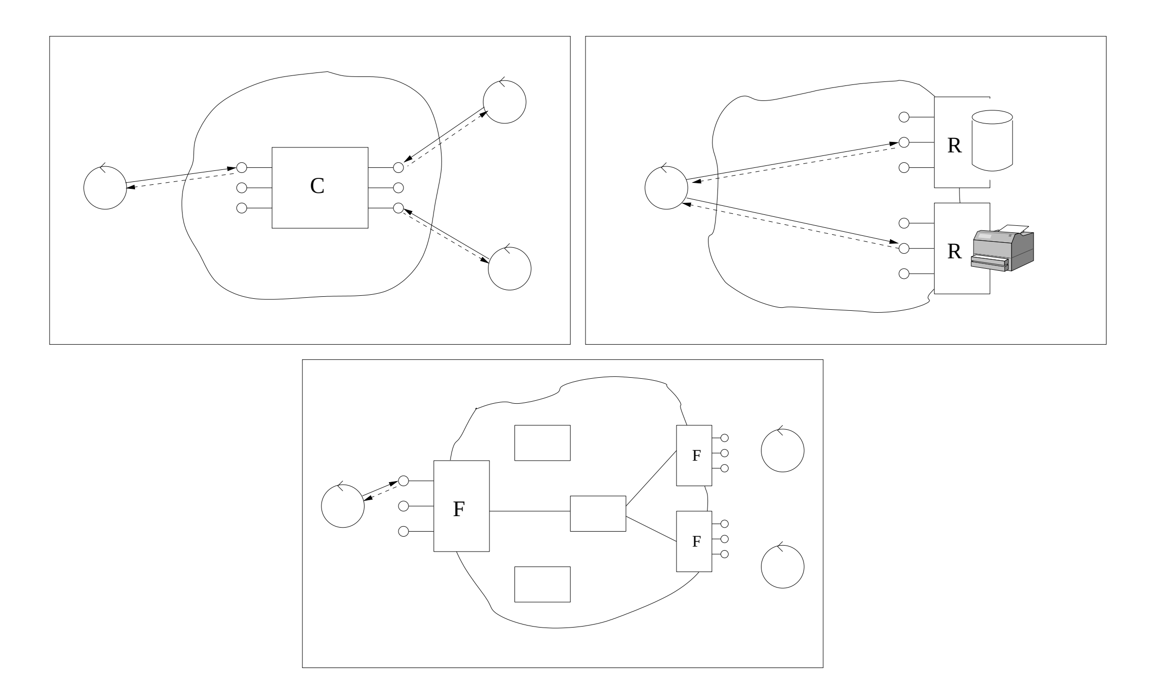


FIG. 3.3 – Différents types d'artifacts : artifacts de coordination (C), artifacts de ressource (R), artifacts frontière (F)

Les artifacts ont d'abord été utilisés pour la coordination entre les agents (artifacts de type C sur la figure). Les artifacts de coordination peuvent être utilisés à différents niveaux d'abstraction pour

[7] Dans une taxonomie, les catégories n'ont pas d'intersection et donc un objet ne peut appartenir qu'à une unique catégorie.

gérer des fonctions de communication (blackboard, boîtes de messages, services de gestion d'événements), des fonctions de synchronisation (ordonnancement, sémaphores, ...) ou des fonctions de coordination de haut niveau (systèmes d'enchères, de workflow, d'infrastructures de phéromones, ...). [Viroli et Ricci, 2004] montre par exemple comment utiliser un artifact pour encapsuler l'implémentation du protocole Contract-Net. Les agents n'ont pas à se préoccuper de toute la mécanique interne du protocole. L'initiateur demande à l'artifact d'initier le protocole en lui indiquant le contenu du contrat. Il reçoit des rapports lui indiquant les propositions faites. L'artifact gère les messages d'acceptation ou de refus envoyés à chacun des participants. [Ricci *et al.*, 2005] donne un autre exemple avec le problème des philosophes, initialement proposé et résolu par Edsger Dijkstra grâce aux sémaphores. L'artifact représente la table à laquelle sont assis les philosophes. Elle permet de coordonner les agents philosophes pour l'accès aux fourchettes. L'implémentation est donnée en ReSpecT et en 3APL.

Un artifact frontière (F) est utilisé pour caractériser et contrôler la présence d'un agent dans un contexte organisationnel, en réifiant un contrat entre un agent et une organisation. Dans les environnements basés sur les rôles, un artifact frontière embarque le contrat pour les rôles que l'agent joue dans l'organisation. Il est fourni à l'agent quand celui-ci démarre une session de travail dans l'organisation. Il permet de contraindre ce que l'agent peut faire dans l'organisation. Il peut être vu comme une interface frontière entre l'agent et l'environnement.

Les artifacts de ressource (R) sont des moyens de médiation d'accès aux ressources physiques ou pour donner une existence à une partie de l'environnement du SMA correspondant à une ressource partageable. Ce type d'artifact est important pour apporter au niveau d'abstraction des agents toutes les entités physiques et de calcul qui peuvent être utilisées par les agents. Les artifacts de ressource remplacent avantageusement les agents *wrappers*[8] couramment utilisés pour résoudre ce problème.

3.3 Les artifacts de calcul

Je présente maintenant la notion d'artifact de calcul, que j'introduis comme une spécialisation de la notion d'artifact. Contrairement aux premiers travaux de l'équipe de Bologne sur les artifacts, qui portaient principalement sur les possibilités de coordination entre agents au travers des artifacts, je m'intéresse dans cette partie aux artifacts comme des outils de calculs disponibles dans l'environnement et utilisés par les agents pour atteindre leurs buts.

3.3.1 Utilité

Les artifacts de calcul vont nous permettre d'externaliser les tâches de calcul des agents pour qu'ils puissent conserver un comportement extraverti. Nous réifions la notion d'outil de calcul dans le but de définir des mécanismes généraux d'utilisation de tels outils, de réduire le fossé qui existe entre les concepts et l'implémentation et enfin de pouvoir permettre la réutilisation des outils de calculs ainsi définis dans différents contextes.

[8]Un agent *wrapper* encapsule l'accès à un système logiciel non agent (un SGBDR par exemple) pour qu'il soit accessible pour les entités du SMA comme s'il faisait partie intégrante de celui-ci.

3.3.2 Définition

Un artifact de calcul est un artifact chargé d'encapsuler l'exécution d'un algorithme, au sens où nous l'avons défini dans la section 2.2.1.1.

Un artifact de calcul dispose au minimum des actions suivantes :
- `start_computation(DATA)` qui permet de démarrer une tâche de calcul,
- `stop_computation` qui permet de stopper la tâche de calcul en cours,
- `pause_computation` qui permet de mettre en pause la tâche de calcul en cours,
- `resume_computation` qui permet de redémarrer la tâche de calcul où elle s'était arrêtée lors de l'appel à `pause_computation`,
- `pause` qui permet de mettre en pause l'artifact,
- `resume` qui permet de réveiller l'artifact.

Nous faisons une distinction entre `pause` et `pause_computation` ainsi qu'entre `resume` et `resume_computation`. `pause` et `resume` sont des opérations qui peuvent être traitées au niveau système sans qu'il y ait besoin que l'artifact s'en rende compte, tandis que `pause_computation` et `resume_computation` doivent être gérées par l'artifact lui-même. Il n'y a aucune différence conceptuelle entre ces deux couples d'actions. Cette distinction est uniquement apportée par notre souci de rester proche de l'implémentation. Des explications plus détaillées sur la nécessité de cette distinction sont données dans le paragraphe suivant.

Les perceptions minimales sont :
- `finished(RESULT)`. Cette perception est reçue lorsque le travail de calcul initié par `start_computation(DATA)` a été terminé avec succès. `RESULT` est unifié avec la résultat donné par l'algorithme lancé.
- `error(ERROR_INFO)`. Cette perception permet de transmettre les erreurs d'exécution de l'algorithme à l'agent.
- `paused`. Cette perception est émise au moment où l'artifact se met en pause.
- `resumed`. Cette perception permet de confirmer le redémarrage de l'artifact.

Les actions et perceptions précédentes constituent une base minimale qu'il est possible d'étendre en fonction du type particulier d'artifact de calcul auquel on a affaire. Par exemple, pour un artifact qui encapsule un algorithme à contrat, on introduira deux nouvelles actions `start_computation(DATA, CONTRACT)` et `continue(CONTRACT)` qui spécifient toutes les deux la durée du contrat. La section 3.4.4 détaille l'instruction opératoire utilisée pour manipuler un algorithme à contrat, ainsi que d'autres exemples d'instructions opératoires classiquement utilisées avec les artifacts de calcul.

3.3.3 Notes d'implémentation et contraintes supplémentaires

Nous présentons dans ce chapitre les concepts que nous voulons mettre en place dans nos systèmes, la mise en œuvre effective étant présentée dans le chapitre 5. Néanmoins, notre but est de

réduire le fossé entre les concepts et l'implémentation. Et comme le concept d'artifact de calcul a des implications très proches du niveau système, nous préférons tout de suite les identifier et adapter le concept en conséquence.

Tout d'abord, nous avons décidé d'externaliser les tâches de calcul afin de simplifier la programmation des agents. Il faut éviter de reporter la complexité de réalisation sur les artifacts de calcul. Nous prendrons donc comme hypothèse qu'un artifact de calcul « appartient » à l'agent qui l'utilise. Les autres agents n'en ont pas connaissance et ne peuvent donc pas tenter de lui faire exécuter des actions pendant qu'il est en train de réaliser une tâche. De plus, un artifact de calcul ne pourra réaliser qu'une tâche de calcul à la fois. Cette contrainte pourra être supprimée par la suite. Dans un premier temps, si l'on veut pouvoir avoir l'illusion qu'un artifact de calcul est partagé par plusieurs agents, il faudra recourir à un artifact de coordination ou de ressource intermédiaire.

Les agents doivent disposer de moyens de démarrer et d'arrêter des artifacts. Ces opérations ne font pas partie de l'interface d'usage des artifacts car ce sont des fonctionnalités de plus haut niveau d'abstraction. L'arrêt brutal d'un artifact peut être demandé à tout moment par un agent. Cela correspond à tuer le processus ou le processus léger qui exécute le programme de l'artifact. L'arrêt normal et non forcé d'un artifact pourra cependant être prévu dans l'interface d'usage de celui-ci.

La mise en pause et le redémarrage des artifacts est nécessaire pour pouvoir mettre en place des mécanismes de partage du processeur comme nous le verrons dans le chapitre suivant. La mise en pause se réalise idéalement au niveau système, sans que ce soit à l'artifact de gérer cela. Sous Unix, on peut par exemple utiliser le signal `SIGSTOP` pour mettre en pause un processus et le signal `SIGCONT` pour le redémarrer. Il faut faire attention que si le processus de l'artifact est mis en pause, il devient impossible pour un agent d'envoyer une commande d'exécution d'une action et d'inspecter les attributs de l'artifact. C'est pourquoi nous proposons un second mécanisme avec les opérations `pause_computation` et `resume_computation`. La mise en pause peut consister à l'arrêt momentané de la tâche de calcul et à l'attente sur une *socket* de la commande de redémarrage. Avec ce système, on peut toujours envoyer d'autres commandes à l'artifact au travers de la *socket* notamment pour inspecter ses attributs. Cette solution possède cependant l'inconvénient de rendre plus difficile le codage de l'artifact, en particulier quand on veut encapsuler du code hérité. En effet, dans la première solution, la gestion des signaux systèmes est entièrement à la charge du système et ce système fonctionne sur n'importe quel programme sans avoir à apporter de modification au code. Dans la seconde solution, il faudra au moins recompiler le code hérité en y ajoutant la gestion des commandes de pause, qui sera entièrement à la charge du processus de l'artifact.

3.4 Le langage de mode d'emploi des artifacts

Nous détaillons maintenant le langage formel qui est utilisé pour décrire les instructions opératoires. Nous décrivons tout d'abord le formalisme à base d'algèbre de processus introduit dans [Viroli et Ricci, 2004]. Ensuite, nous présentons les extensions que nous y avons introduit, nous faisons une analyse critique permettant de mettre en évidence les lacunes et nous présentons de manière plus détaillée la manière d'exprimer le mode d'emploi des artifacts de calcul.

3.4.1 Les algèbres de processus

Milner a initié l'étude des systèmes concurrents communicants [Milner, 1982]. Il a voulu mettre en évidence une description abstraite des mécanismes fondamentaux de la concurrence ainsi qu'un calcul pour raisonner sur les programmes concurrents. A sa suite, s'est développée une approche algébraico-axiomatique de la théorie de la concurrence, connue sous le nom d'algèbre de processus.

La différence entre les travaux initiaux de Milner et les algèbres de processus actuelles réside dans l'approche suivie pour décrire la concurrence. Dans les premiers travaux, tous les processus sont synchronisés sur une horloge universelle et on se donne des outils pour rompre cette organisation de manière à pouvoir exprimer l'asynchronisme. Dans les algèbres de processus, les processus sont vus comme des entités « autonomes » donc asynchrones, et l'on se donne les moyens d'organiser la cohérence du système de processus.

Un processus p est conçu abstraitement comme un objet qui accomplit certaines actions a et ce faisant se reconfigure en un autre processus p'. Cette relation est dénotée par :

$$p \xrightarrow{a} p'$$

Un processus se termine avec succès (état $\surd$) s'il n'est plus constitué que d'une action atomique et que celle-ci est exécutée :

$$a \xrightarrow{a} \surd$$

La plupart des algèbres de processus contiennent des opérateurs de base pour construire des processus finis, des opérateurs de communication pour exprimer la concurrence, et une certaine notion de récursion pour représenter les comportements infinis.

Les algèbres de processus permettent un raisonnement formel sur les processus et les données et peuvent être utilisés pour faire de la vérification des systèmes. On pourra se reporter à [Fokkink, 2000] pour plus d'informations théoriques et pratiques sur les algèbres de processus.

3.4.2 Le langage initial

Nous allons maintenant décrire le formalisme à base d'algèbre de processus proposé dans [Viroli et Ricci, 2004] pour décrire les instructions opératoires des artifacts.

Soit Δ l'ensemble des actes d'interaction possibles et $\mathcal{I}$ l'ensemble des instructions opératoires. Les définitions d'un acte d'interaction $\delta \in \Delta$ et d'une instruction opératoire $I \in \mathcal{I}$ sont :

$$\delta ::= \alpha \mid \pi$$

$$I ::= 0 \mid !\alpha \mid \underline{\alpha} \mid ?\pi \mid I; I \mid I + I \mid (I || I) \mid \mathcal{D}(t_1, \ldots, t_n)$$

Un acte d'interaction δ peut être une action α ou une perception π. Une instruction I peut être une instruction atomique : le comportement nul (0), le déclenchement de l'exécution de l'action α ($!\alpha$), une action débutée mais pas encore terminée ($\underline{\alpha}$) et la perception π de la terminaison d'une action ($?\pi$). Une instruction peut également être structurée grâce à différents opérateurs : « ; » pour la composition séquentielle de deux instructions, « + » pour le choix et « $||$ » pour la composition parallèle. Enfin, pour permettre la récursion, une instruction peut être l'invocation $\mathcal{D}(t_1, \ldots, t_n)$ d'une autre instruction opératoire avec $\mathcal{D}$ le nom de l'instruction opératoire à appeler et $t_1, \ldots, t_n$ la liste de ses paramètres.

Des règles de congruence sont définies pour exprimer la commutativité et la transitivité des différents opérateurs, de définir le comportement de l'instruction vide et de l'invocation d'une autre instruction opératoire :

$$
\begin{array}{ccc}
0 + I \equiv I + 0 & I + I' \equiv I' + I & I + (I' + I'') \equiv (I + I') + I'' \\
0||I \equiv I||0 & I||I' \equiv I'||I & I||(I'||I'') \equiv (I||I')||I'' \\
0; I \equiv I & I; 0 \equiv I & (I + I'); I'' \equiv (I; I') + (I'; I'')
\end{array}
$$

$$
\mathcal{D}(t_1, \ldots, t_n) \equiv I[t_1/v_1, \ldots, t_n/v_n] \qquad si \qquad \mathcal{D}(v_1, \ldots, v_n) := I
$$

Il est possible de donner une sémantique opérationnelle à ce langage. Dans les règles suivantes, la notation $I \xrightarrow{\delta}_{\mathcal{I}} I'$ signifie que l'état de l'instruction opératoire $I \in \mathcal{I}$ se transforme en l'état $I' \in \mathcal{I}$ par l'occurrence de l'acte d'interaction $\delta \in \Delta$. La notation $[t/t']$ est utilisée pour indiquer la substitution de t par t' dans la suite des instructions.

$$
\begin{array}{lll}
I||I'' \xrightarrow{\delta}_{\mathcal{I}} I'||I'' & \text{si } I \xrightarrow{\delta}_{\mathcal{I}} I' & \text{[PAR]} \\
I + I' \xrightarrow{\delta}_{\mathcal{I}} I'' & \text{si } I \xrightarrow{\delta}_{\mathcal{I}} I'' \text{ et } I' \not\xrightarrow{\delta}_{\mathcal{I}} I''' & \text{[ECH]} \\
I + I' \xrightarrow{\delta}_{\mathcal{I}} I'' + I''' & \text{si } I \xrightarrow{\delta}_{\mathcal{I}} I'' \text{ et } I' \xrightarrow{\delta}_{\mathcal{I}} I''' & \text{[LCH]} \\
!\alpha; I \xrightarrow{\alpha'}_{\mathcal{I}} \underline{\alpha'}; [\alpha/\alpha']I & & \text{[ACT]} \\
\underline{\alpha}; ((?\pi; I) + I'); I'' \xrightarrow{\alpha:\pi'}_{\mathcal{I}} [\pi/\pi'](I; I'') & & \text{[COMP]}
\end{array}
$$

La règle [PAR] spécifie qu'une seule instruction opératoire est suivie à un instant donné. La règle [ECH] définit la sémantique du choix exclusif : si I est exécutée, les choix à l'intérieur de I' sont exclus. La règle [LCH] au contraire définit le choix retardé : si l'interaction δ permet plusieurs choix, δ est exécuté et le choix est retardé à une prochaine interaction. La règle [ACT] spécifie que si α est la prochaine action à exécuter, une spécialisation α' de α peut être exécutée, propageant la substitution correspondante dans la suite des instructions. La règle [COMP] traite de manière similaire la perception de la terminaison d'une action.

Ces règles permettent de construire un automate dont les états sont les différentes formes que peut prendre une instruction opératoire durant son exécution. L'état initial de l'automate correspond à l'état de départ de l'instruction opératoire lorsqu'elle est instanciée. L'état final S correspond à la terminaison de l'exécution de l'instruction opératoire avec succès ($\surd$). Les transitions entre les états correspondent à l'activation des règles de la sémantique opérationnelle.

Prenons comme exemple les deux instructions opératoires suivantes :

```
IO1 := (?a1 ; !p1) +  (?a2 ; !p2)
IO2 := (?a3 ; !p3) || (?a4 ; !p4)
```

Nous pouvons construire à partir de ces instructions opératoires les automates de la figure 3.4, qui donnent l'ensemble des chemins d'exécution possibles de ces instructions opératoires. L'automate de gauche correspond à `OI1` et celui de droite à `IO2`.

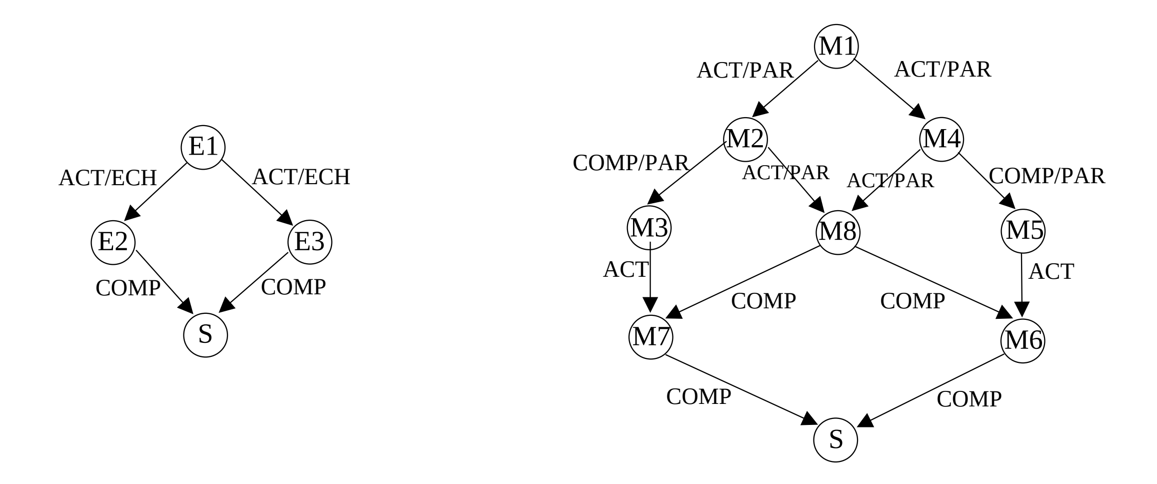


FIG. 3.4 – Exemple d'automates sous-jacents à des instructions opératoires.

Voici la correspondance entre les états des automates de la figure et les différentes valeurs que peuvent prendre les instructions opératoires durant leur exécution (nous avons considéré ici un choix exclusif entre `a1` et `a2`) :

État de l'automate	Instruction opératoire
E1	(?a1;!p1) + (?a2;!p2)
E2	(<u>a1</u>;!p1)
E3	(<u>a2</u>;!p2)
M1	(?a3;!p3) || (?a4;!p4)
M2	(<u>a3</u>;!p3) || (?a4;!p4)
M3	?a4;!p4
M4	(?a3;!p3) || (<u>a4</u>;!p4)
M5	?a3;!p3
M6	<u>a3</u>;!p3
M7	<u>a4</u>;!p4
M8	(<u>a3</u>;!p3) || (<u>a4</u>;!p4)

Viroli et Ricci ont proposé dans [Viroli et Ricci, 2004] une extension à l'algèbre précédente pour prendre en compte la notion de timeout dans les instructions opératoires. Ils définissent un timeout ainsi :

Définition 3.1 (Timeout) *Un timeout définit une période de temps à la fin de laquelle une action peut être associée.*

Un timeout peut être utilisé pour se mettre en attente pendant une durée maximale d'un événement. Il peut également être utilisé pour réaliser une action pendant une durée maximale. Un timeout peut être désactivé à tout instant avant l'expiration de sa période de temps. Dans ce cas, l'action associée à l'expiration de la période de temps n'est pas exécutée.

L'extension pour gérer les timeouts est obtenue en étendant les définitions des instructions et des actes d'interaction de la manière suivante :

$$I ::= \ldots \mid T(n) \mid \epsilon \mid \gamma \qquad\qquad \delta ::= \ldots \mid \tau(m)$$

$T(n)$ est l'instruction timeout, le paramètre n étant la durée du timeout.$\tau(m)$ est la perception de m unités de temps. ϵ est un état d'erreur dans lequel l'instruction opératoire tombe si un timeout a expiré pendant qu'elle était en attente du lancement d'une action. Cela correspond à une faute de l'agent qui n'a pas respecté le mode d'emploi de l'artifact. γ est l'état d'erreur dans lequel on se retrouve si un timeout expire alors qu'on était en attente d'une perception. Cet état ne doit normalement jamais être atteint car on considère que les artifacts n'ont aucune latitude par rapport à leurs spécifications : ils respectent obligatoirement leur mode d'emploi.

Comme pour les instructions précédentes, les règles suivantes donnent une sémantique opérationnelle à l'instruction timeout :

$$\begin{array}{lll}
T(n); I \xrightarrow{\tau(m)}_{\mathcal{I}} T(n-m); I & \text{si } n > m & \text{[T-A-PASS]} \\
T(m); I \xrightarrow{\tau(n)}_{\mathcal{I}} \epsilon & \text{si } n \geq m & \text{[T-A-EXP]} \\
\underline{\alpha}; T(n); I \xrightarrow{\tau(m)}_{\mathcal{I}} \underline{\alpha}; T(n-m); I & \text{si } n > m & \text{[T-P-PASS]} \\
\underline{\alpha}; T(m); I \xrightarrow{\tau(n)}_{\mathcal{I}} \gamma & \text{si } n \geq m & \text{[T-P-EXP]} \\
T(n); !\alpha; I \xrightarrow{\alpha'}_{\mathcal{I}} \underline{\alpha'}; [\alpha/\alpha']I & & \text{[T-ACT]} \\
\underline{\alpha}; T(n); ((\pi; I) + I'); I'' \xrightarrow{\alpha:\pi'}_{\mathcal{I}} [\pi/\pi'](I; I'') & & \text{[T-COMP]}
\end{array}$$

La règle [T-A-PASS] gère le passage du temps pendant le temps où le timeout n'a pas encore expiré et qu'une action doit être réalisée. Quand le timeout expire, la règle [T-A-EXP] fait passer l'instruction opératoire dans l'état d'erreur ϵ. L'agent doit tout faire pour ne pas se retrouver en situation d'échec dans l'utilisation de l'artifact. Il doit donc interpréter la règle [T-A-EXP] pour éviter l'état d'erreur ϵ : il le fait en exécutant l'action attendue avant l'expiration du timeout.

Les règles [T-P-PASS] et [T-P-EXP] ont la même fonction mais quand une perception est attendue : [T-P-PASS] gère le passage du temps et [T-P-EXP] fait tomber l'instruction opératoire dans l'état d'erreur γ quand le timeout expire. Les artifacts étant conçus pour respecter à la lettre leur mode

d'emploi, il n'y a aucune raison pour que l'état d'erreur γ soit atteint. L'agent peut donc interpréter ces règles dans le sens où il dispose de la garantie que la perception qu'il attend se produira avant l'expiration du timeout.

La règle [T-ACT] supprime le timeout quand l'agent décide d'exécuter la spécialisation α' de l'action α. La règle [T-COMP] fait de même quand une perception est réalisée avant que le timeout expire.

3.4.3 Nos extensions

Le modèle théorique pour décrire les modes d'emploi des artifacts à base d'algèbre de processus nous paraît incomplet. Nous pensons que cette base de travail est peut-être trop rigide pour pouvoir décrire des modes d'emploi complexes et flexibles[9]. Nous avons cependant décidé de travailler dans un premier temps avec ce modèle, en attendant de trouver un autre langage qui nous conviendra mieux. Nous avons identifié un certain nombre de manquements au langage de base fourni par l'équipe italienne. Il est possible de palier quelques-uns d'entre eux grâce aux possibilités d'extension faciles à mettre en œuvre dans les algèbres de processus. C'est ce que nous décrivons dans cette section. Nous terminons en décrivant les fonctionnalités plus difficiles à mettre en place dans le modèle actuel.

3.4.3.1 L'attente

Comme nous le verrons dans le chapitre suivant quand nous voudrons utiliser plusieurs artifacts sur le même processeur, nous avons besoin de pouvoir mettre en pause et redémarrer les artifacts aux limites d'intervalles qui seront donnés par un ordonnanceur.

La gestion des timeouts décrite précédemment signifie intuitivement que les agents ont une action à réaliser ou qu'une perception arrivera *avant* une date donnée. Cela peut permettre d'indiquer à l'agent de mettre en pause l'artifact avant la fin de l'intervalle qui lui est imparti.

Nous avons aussi besoin de la possibilité d'indiquer à un agent qu'il doit faire une action *après* une date donnée. Tout comme pour le timeout, nous étendons donc la définition d'une instruction avec la nouvelle instruction atomique $W(n)$ et nous ajoutons deux règles opérationnelles :

$$I ::= \ldots \mid W(n)$$

$$W(n); I \xrightarrow{\tau(m)}_{\mathcal{I}} W(n-m); I \quad \text{si } n > m \quad \text{[W-T-PASS]}$$

$$W(m); I \xrightarrow{\tau(n)}_{\mathcal{I}} I \quad \text{si } n > m \quad \text{[W-T-EXP]}$$

La règle [W-T-PASS] gère le passage du temps pendant que l'attente n'est pas finie. La règle [W-T-EXP] gère l'expiration du délai imposé avant de pouvoir exécuter la prochaine instruction. Notons

[9] voir section 3.4.3.5 pour plus de détails.

qu'il n'y a pas de règle [W-ACT]. Dans le cas du timeout, une action peut être réalisée avant que le timeout n'expire, grâce à la règle [T-ACT]. Ici, l'agent ne peut pas démarrer une action avant la fin de l'attente.

Pour clarifier ces notions, voici l'exemple d'une instruction opératoire qui peut être utilisée pour contrôler et interagir avec un artifact de calcul :

```
OI := W(10); !start(data);
  ((T(5); !pause; W(10); !restart; T(5); !stop)
    + ?finished(result)
    + ?failure(error_info))
```

L'agent doit attendre 10 tops d'horloge avant d'ordonner à l'artifact de démarrer sa tâche de calcul avec `DATA` comme données d'entrée. Il doit ordonner à l'artifact de se mettre en pause moins de 5 tops d'horloge après le démarrage de la tâche et de redémarrer pas moins de 10 tops d'horloge plus tard. Enfin, si la tâche de calcul ne s'est pas terminée - correctement ou à la suite d'une erreur - en moins de 5 tops d'horloge (instruction de timeout), l'agent doit lui ordonner de mettre un terme à son calcul courant (action `stop`).

3.4.3.2 L'alternative après un timeout

Les états d'erreurs introduits pour gérer l'expiration d'un timeout posent problème car une fois qu'on les a atteint, il n'est plus possible d'en ressortir. D'un côté, il est important que l'artifact respecte son mode d'emploi, mais d'un autre côté, il serait intéressant de spécifier ce qui se passe lorsque c'est l'agent qui a des obligations qu'il ne respecte pas. Par exemple, si l'agent dispose de deux secondes pour exécuter une action et qu'il ne le fait pas, l'artifact ne va pas forcément cesser d'exister ou de fonctionner. On aimerait aussi pouvoir spécifier des comportements de l'artifact un peu plus complexes comme : « l'artifact répond normalement en deux secondes ; sinon, il donnera une réponse partielle une seconde plus tard ».

Paul-Édouard Marson a proposé durant son Master Recherche une généralisation de l'instruction de timeout permettant d'améliorer la gestion des cas où l'agent ne respecte pas les délais qui lui sont imposés [Marson, 2006].

Nous ajoutons une nouvelle instruction $T(n, R)$ où R est une instruction opératoire et les règles opérationnelles suivantes :

$$I ::= \dots \mid T(n, R)$$

$T(n,R); I \xrightarrow{\tau(m)}_{\mathcal{I}} T(n-m,R); I$	si $n > m$	[A-PASS-EXT]
$T(m,R); I \xrightarrow{\tau(n)}_{\mathcal{I}} R$	si $n \geq m$	[A-VIO-EXT]
$\underline{\alpha}; T(n,R); I \xrightarrow{\tau(m)}_{\mathcal{I}} \underline{\alpha}; T(n-m,R); I$	si $n > m$	[P-PASS-EXT]
$\underline{\alpha}; T(m,R); I \xrightarrow{\tau(n)}_{\mathcal{I}} \gamma$	si $n \geq m$	[P-EXP-EXT]
$T(n,R); !\alpha; I \xrightarrow{\alpha'}_{\mathcal{I}} \underline{\alpha'}; [\alpha/\alpha']I$		[T-ACT-EXT]
$\underline{\alpha}; T(n,R); ((\pi; I) + I'); I'' \xrightarrow{\alpha:\pi'}_{\mathcal{I}} [\pi/\pi'](I; I'')$		[T-COMP-EXT]

La règle [A-VIO-EXT] spécifie que si un agent ne respecte pas les temporisations définies dans les instructions opératoires, il ne tombe pas dans un état d'erreur mais est autorisé à suivre les instructions opératoires R. Les autres règles sont calquées sur celles présentées dans la section 3.4.2. En particulier, la règle [P-EXP-EXT] indique que l'on tombe bien dans l'état d'erreur γ quand une perception n'arrive pas à temps : l'artifact reste obligé de respecter son mode d'emploi.

Nous pouvons remarquer que $T(n, R)$ est une généralisation de $T(n)$. En effet, $T(n)$ subsume $T(n, R)$ avec $R = \epsilon$.

3.4.3.3 Utilisation de l'historique

Comme nous l'avions déjà souligné dans le cas de TÆMS, l'utilisation de boucles avec des structures arborescentes n'est pas aisée.

Nous aimerions cependant pouvoir exprimer dans les modes d'emploi de nos artifacts des utilisations répétitives de certaines fonctionnalités et également pouvoir exprimer des conditions d'utilisation en fonction de l'utilisation passée qui a été faite de l'artifact.

Par exemple, concernant le mode d'emploi d'un lave-vaisselle, il serait intéressant de pouvoir y spécifier qu'« il faut remplir le bac à sel tous les dix lavages ». Cela peut être réalisé en utilisant un état de l'instruction opératoire par lavage : après le premier lavage, on se retrouve dans un état identique au précédent, mais correspondant au second lavage. Après le dixième lavage, il n'est plus possible de remettre en route un nouveau lavage. Il faut remplir le bac à sel et ensuite on revient dans l'état initial pour un nouveau cycle de dix lavages. Ce fonctionnement est schématisé dans la figure 3.5. L'instruction opératoire correspondante est la suivante :

```
lave_vaisselle_simple :=
  (  (!laver ; ?fin_lavage) ; (!laver ; ?fin_lavage)  ;
     ... ; (!laver ; ?fin_lavage)
     ; (!remettre_du_sel ; ?fin_remplissage)
     ; lave_vaisselle_simple
```

Il est bien évident qu'il ne sera possible de gérer ainsi que des boucles constituées de quelques itérations. Nous proposons d'utiliser plutôt des informations sur l'historique des états de l'instruction opératoire pour inhiber ou désinhiber des branches de celui-ci. L'historique est vu comme la

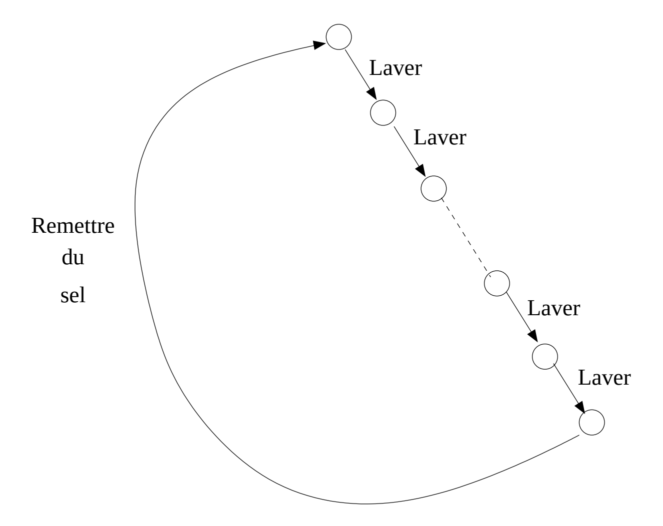


FIG. 3.5 – Mode d'emploi d'un lave-vaisselle qui ne prend pas en compte l'historique.

succession des événements qui se sont produits : lancement d'une action (`a_nom_action`) ou déclenchement d'une perception (`p_nom_perception`). La succession des événements est conservée comme ceci :
`H=«a_action1;p_perception1;a_action2;p_perception2;»`

Les conditions sur l'historique sont exprimées sous la forme d'expressions rationnelles[10]. On dispose des opérateurs suivants :

- `.` pour une occurence unique d'une action ou d'une perception quelconque,
- `a_.` pour une occurence unique d'une action quelconque,
- `p_.` pour une occurence unique d'une perception quelconque,
- `*` pour zéro ou plusieurs occurences de ce qui précède,
- `?` pour zéro ou une occurence de ce qui précède,
- `+` pour une ou plusieurs occurences de ce qui précède,
- `{n}` pour une répétition `n` fois de ce qui précède,
- `;` pour la succession,
- `|` pour le choix,
- `^` pour une correspondance avec le début de l'historique,
- `$` pour une correspondance avec la fin de l'historique.

Au moment de chaque transition dans l'instruction opératoire, les conditions pour chacun des nœuds sont évaluées. L'évaluation est faite en commençant par la fin de l'historique pour identifier l'emplacement le plus récent de la condition. Lorsqu'une condition est validée, les actions composées correspondantes sont exécutées. On dispose de deux actions primitives qu'il est possible d'utiliser pour masquer (`deactivate`) ou réactiver (`activate`) des actions ou perceptions possibles. C'est l'instruction opératoire finalement obtenue après application de toutes les règles qui est utilisée par l'agent pour prendre la décision de la conduite à tenir dans l'état courant.

[10] *regular expression* ou *regex* en anglais.

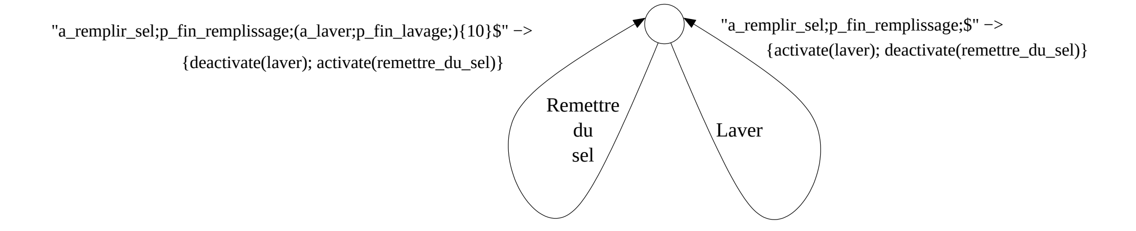


FIG. 3.6 – Mode d'emploi d'un lave-vaisselle prenant en compte l'historique.

Pour notre exemple de mode d'emploi d'un lave-vaisselle, la figure 3.6 illustre le mode d'emploi que nous obtenons en utilisant l'historique. L'instruction opératoire est la suivante :

```
lave_vaisselle_historique :=
  (  (!laver ; ?fin_lavage) + (!remettre_du_sel ; ?fin_remplissage)
     ; lave_vaisselle_historique
```

La règle

```
«a_remettre_du_sel;p_fin_remplissage;$» ->
    {activate(laver);deactivate(remettre_du_sel)}
```

permet de désactiver le remplissage du sel et d'autoriser de nouveau les lavages lorsque la toute dernière action a consisté à remettre du sel. Le `$` permet d'indiquer une correspondance avec la fin de l'historique.

La règle

```
«a_remettre_du_sel;p_fin_remplissage;(a_laver;p_fin_lavage;){10}$» ->
    {deactivate(laver);activate(remettre_du_sel)}
```

permet quant à elle de faire l'inverse lorsque l'on a détecté dix cycles de lavages après un remplissage du sel.

Ce mécanisme permet d'introduire de manière simple des boucles dans nos instructions opératoires. Nous étudions actuellement comment flexibiliser celui-ci en ne conservant dans l'historique que certaines informations ou pour pouvoir purger les parties les plus anciennes de l'historique qui ne sont plus nécessaires.

Remarquons que l'introduction de ce mécanisme d'inhibition de nœuds dans l'instruction opératoire limite considérablement les possibilités de raisonnement que l'agent peut effectuer sur l'instruction opératoire, ceci car elle n'est plus fixe. Le plus simple est d'utiliser dans l'agent un comportement réactif qui réévalue à chaque étape la bonne action à réaliser. C'est ce que nous faisons actuellement dans nos systèmes (voir section 5.1.3).

3.4.3.4 Possibilités de raisonnement pour l'agent

3.4.3.4.1 Lien sémantique avec la base de connaissances de l'agent

Les instructions opératoires, données seules, ne sont d'aucune utilité aux agents. Il faut leur donner des informations supplémentaires leur permettant de décider des actions qu'ils vont lancer sur les artifacts et de comprendre les perceptions qu'ils vont recevoir en retour.

On utilise pour cela une sémantique mentale associée aux actions et perceptions [Viroli et Ricci, 2004]. Elle est définie dans une table qui indique pour chaque action l'état mental dans lequel l'agent doit être pour décider de lancer cette action et pour chaque perception les modifications à apporter à l'état mental.

Prenons l'exemple d'un mode d'emploi rudimentaire pour un lave-linge. On peut lancer un cycle de lavage et percevoir sa terminaison puis recommencer autant de fois que l'on veut. L'instruction opératoire correspondante est la suivante :

```
laver_linge :=
  !lancer_cycle ; ?fin_cycle ; laver_linge
```

Nous voulons pouvoir indiquer à l'agent quand il pourra décider d'exécuter l'action `lancer_cycle`. Cela s'exprime ainsi :

Action	Précondition	Perception	Effet
lancer_cycle	B linge_sale	fin_cycle	B ¬linge_sale

L'agent décidera d'exécuter l'action `lancer_cycle` s'il croit (opérateur `B`) que son linge est sale. Les effets liés à la perception `fin_cycle` permettent de mettre à jour la base de connaissances pour que l'agent sache maintenant que son linge est propre.

Nous pouvons également utiliser les paramètres des actions et perceptions dans cette table. Prenons par exemple le cas d'un agent qui aurait besoin de connaître différentes informations détenues par un artifact. L'instruction opératoire qui lui permet de le faire est la suivante :

```
recuperer_info :=
  !demande_info(NOM_INFO) ; ?valeur_info(NOM_INFO, VALEUR)
```

Le lien sémantique entre cette instruction opératoire et la base de connaissances de l'agent permet d'indiquer à l'agent qu'il peut lancer l'action `demande_info(NOM_INFO)` lorsqu'il a besoin de connaître la valeur de l'information dont le nom est `NOM_INFO` :

Action	Précondition
demande_info(NOM_INFO)	B besoin_info(NOM_INFO)

Perception	Effet
valeur_info(NOM_INFO, VALEUR)	B valeur(NOM_INFO, VALEUR) B ¬besoin_info(NOM_INFO)

Nous utilisons ce lien sémantique dans le chapitre suivant et on pourra trouver un exemple concret d'utilisation dans la section 4.1.7.2.

Cette solution est simple et fonctionnelle mais elle oblige à connaître au moment de la mise au point de l'artifact la structuration de la base de croyances des agents. De plus, elle impose un fonctionnement très procédural où, pour décider de la prochaine action à réaliser, l'agent a juste à consulter sa base de connaissances.

3.4.3.4.2 Utilisation de Design-to-Criteria

Pour palier les limitations du modèle précédent, nous proposons une méthode qui permet aux agents d'utiliser les possibilités de raisonnement et d'ordonnancement offerts par Design-to-Criteria, déjà présenté dans la section 2.2.3.4.4. Design-to-Criteria prend en entrée un arbre de tâches exprimé dans le formalisme de TÆMS. Nous proposons donc une méthode permettant de traduire une instruction opératoire exprimée dans notre langage à base d'algèbre de processus en un arbre TÆMS.

Dans notre algèbre de processus, les actions et perceptions sont des primitives atomiques et sans durée. La durée des tâches exécutées par les artifacts apparaît grâce à la succession d'une action et de la perception associée. L'utilisation du timeout et de l'attente entre une action et une perception permet d'expliciter la durée des tâches. Dans TÆMS, les *méthodes* représentent les actions primitives, qui peuvent s'inscrire dans la durée.

Nous traduisons une séquence `(!a; T(n);?p)` par une méthode nommée `a` et en indiquant que sa durée minimale est 0 et sa durée maximale est `n`. L'attente `W(n)` sera quant à elle traduite par une méthode `Wait` de durée minimale `n`.

Dans TÆMS, les fonctions d'accumulation de qualité (FAQ) permettent d'indiquer comment est calculée la qualité pour un nœud de l'arbre en fonction de la qualité obtenue par chacun de ses fils. Il en existe 11. Leur sémantique précise est définie dans [Horling, 1999].

L'opérateur parallèle « || » peut être traduit par les FAQ `q_min`, `q_sum` ou `q_all_sum`. L'alternative « + » pourra être traduite par deux FAQ différentes en fonction de son type. Si c'est un choix exclusif, ce sera `q_exactly_one` et si c'est un choix retardé, ce sera `q_max`. L'opérateur de séquence « ; » sera traduit par une des FAQ de séquence : `q_seq_min`, `q_seq_max`, `q_seq_sum` ou `q_seq_last`.

Il n'est pas possible d'automatiser cette traduction car un opérateur de notre algèbre de processus peut être traduit en plusieurs FAQ différentes. Par exemple, la séquence peut être traduite par `q_seq_min` qui spécifie que les sous-tâches doivent être réalisées en séquence et que la tâche prend ensuite la qualité minimale de toutes les qualités obtenues pour les sous-tâches. On peut aussi la traduire par `q_seq_sum`, mais fonctionne de la même manière à part que la qualité obtenue pour la tâche est la somme des qualités de sous-tâches. Le choix entre les différentes traductions possibles dépend du critère choisi pour faire fonctionner Design-to-Criteria et de la manière avec laquelle on veut guider l'algorithme dans les différentes branches de l'arbre.

Cette traduction des instructions opératoires vers TÆMS peut être mise à profit pour autoriser des raisonnements sur les différentes alternatives qui existent dans les instructions opératoires. Notons que nous pouvons toujours utiliser notre extension concernant l'utilisation de l'historique mais que cela impose de réappliquer Design-to-Criteria à chaque fois que l'activation d'une condition sur l'historique a modifié l'instruction opératoire.

3.4.3.5 Fonctionnalités manquantes

Les modes d'emploi tels que nous pouvons les définir actuellement permettent de définir un mode d'emploi qu'il est obligatoire de suivre. Il n'est pas possible d'y indiquer des conseils d'utilisation. Dans l'exemple du lave-vaisselle précédent, on pourrait ainsi indiquer que le remplissage du bac à sel est recommandé tous les dix lavages mais que ceci n'est pas obligatoire.

La gestion du temps sous forme de paramètres de durées dans les instructions d'attente et de timeout est également très limitative. Il serait intéressant de remplacer ces paramètres de durée par des contraintes temporelles. Ces contraintes temporelles pourraient également être plus globales, par exemple pour spécifier qu'un artifact n'est en fonctionnement qu'entre 12 heures et 14 heures.

Nous aimerions également pouvoir disposer dans les instructions opératoires de certaines possibilités du langage TÆMS. La sémantique de l'alternative pourrait par exemple être étendue pour autoriser à changer d'alternative une fois que l'on s'est aperçu que celle que l'on avait choisi auparavant n'aboutit pas. Il faut cependant remarquer que ce type d'extension est au prix d'une plus grande complexité de la partie de l'agent chargée d'analyser le mode d'emploi de l'artifact.

3.4.4 Mode d'emploi des artifacts de calcul

Nous avons vu dans la section 2.2.1 les différentes informations dont on peut disposer sur les algorithmes et quelles sont les différentes manières de les classer.

Ce qui nous intéresse dans cette étude, ce sont les informations qui nous permettent de contrôler les algorithmes. Nous classons les algorithmes ainsi, en fonction de la quantité d'information dont on dispose sur leur comportement temporel :

- **« boîte noire ».** Aucune information n'est connue, à part que c'est un algorithme au sens donné dans la section 2.2.1 et donc une fois lancé, il se terminera au bout d'un temps fini.
- **durée d'exécution connue.** Nous ne disposons que de la durée d'exécution mais pas d'informations sur ce qui se passe entre le début et la fin de l'exécution.
- **à contrat.** L'algorithme peut être contrôlé en lui donnant une durée au bout de laquelle il s'arrête pour fournir le résultat courant. Il est possible de le relancer pour une nouvelle période de temps à partir de l'endroit où il s'était arrêté. On ne dispose pas d'informations sur ce qui se passe pendant chaque période d'exécution. Un agent qui utilise un tel algorithme doit correctement choisir la durée des contrats. Trop grandes, on se retrouve dans le flou trop longtemps ; trop petites, l'agent va devoir reconsidérer trop souvent s'il poursuit ou arrête le calcul.
- **interruptible.** On n'a plus besoin de la notion de contrat car l'algorithme est capable de fournir un résultat à tout moment.
- **anytime.** On dispose d'une information supplémentaire : le profil de la qualité du résultat en fonction du temps alloué à l'exécution de l'algorithme.

On peut considérer que cette classification est également un classement des algorithmes du moins favorable (boîte noire) au plus favorable (anytime). Nous donnons dans les paragraphes suivants les modes d'emploi génériques pour l'encapsulation des algorithmes de cette classification dans des artifacts de calcul.

3.4.4.1 Algorithme « boîte noire »

Le mode d'emploi d'un algorithme pour lequel on ne connaît pas le temps d'exécution est le suivant :

```
black_box_algorithm :=
  !start_computation(INPUT_DATA) ;
    (  (  (?finished(RESULT) + ?error(ERROR_INFO))
           + (!stop_computation ; ?computation_stopped))
       ||  pause_restart) ;
       black_box_algorithm

pause_restart :=
  (  (  (!pause; ?paused; !restart; ?restarted)
        + (!pause_computation; ?paused; !restart_computation; ?restarted))
     ; pause_restart)
   + test_finished

test_finished :=
  ?finished(RESULT) + ?error(ERROR_INFO) + ?computation_stopped
```

Tout d'abord, l'agent n'a qu'une action possible : démarrer le calcul. Il le fait quand bon lui semble. Ensuite, il sait qu'il recevra soit le résultat du calcul soit un rapport d'erreur. Aucun délai n'est garanti pour cela. A tout moment, il a la possibilité d'arrêter le calcul ou d'effectuer des cycles de mise en pause et de redémarrage selon l'un des deux modes proposés. Quand un calcul est terminé, on reboucle pour pouvoir en relancer un nouveau.

L'utilisation de `test_finished` permet de garantir que l'on ne pourra plus tenter de mettre en pause un calcul déjà terminé. Remarquons que nous faisons l'hypothèse que `!pause` est immédiatement suivi de `?paused` sans que `?finished` ou `?error` puisse se produire entre les deux.

3.4.4.2 Algorithme à durée d'exécution connue

On ajoute juste à l'instruction opératoire précédente un timeout qui garantit que l'on aura soit la réponse soit une erreur avant la fin du temps imparti. Cela donne :

```
known_execution_time_algorithm :=
  !start_computation(INPUT_DATA) ;
    (  T(DURATION) ; (  (?finished(RESULT) + ?error(ERROR_INFO))
           + (!stop_computation ; ?computation_stopped))
       ||  pause_restart) ;
       known_execution_time_algorithm
```

Nous considérons ici que la durée d'exécution `DURATION` est un attribut de l'artifact. Elle peut être fixe ou calculée et mise à jour par l'agent avant de lancer un nouveau calcul.

3.4.4.3 Algorithme à contrat

Le mode d'emploi d'un algorithme à contrat est le suivant :

```
 contract_algorithm :=
  !start_computation(INPUT_DATA, CONTRACT) ;
    ( ( T(CONTRACT) ; (  (?finished(RESULT) + ?error(ERROR_INFO))
           + (!stop_computation ; ?computation_stopped)
           + (?end_contract ; contract)))
       ||  pause_restart) ;
       contract_algorithm

contract := !continue(CONTRACT) ;
     ( ( T(CONTRACT) ; (?finished(RESULT) + ?error(ERROR_INFO)))
         + (!stop_computation ; ?computation_stopped)
         + (?end_contract ; contract))
```

La structure de base est la même que précédemment à part de l'on ajoute comme paramètre à `start_computation` la durée du premier contrat. Ensuite, tant que le calcul n'est pas terminé et qu'il n'y a pas eu d'erreur, il est possible de relancer de nouveaux contrats.

3.4.4.4 Algorithme interruptible

Nous avons ici deux manières de faire. La première consiste à prendre la même instruction opératoire que pour le cas d'un algorithme boîte noire et l'artifact exhibe un attribut qui représente le résultat actuel que l'agent peut lire à tout moment. On fera juste attention de redéfinir

l'instruction `pause_restart` pour que l'on ne puisse utiliser que `pause_computation` et `restart_computation`. Cela donne :

```
pause_restart :=
  (  (!pause_computation; ?paused; !restart_computation; ?restarted)
     ; pause_restart)
   + test_finished
```

La seconde méthode consiste à rajouter une action `get_current_result` et une perception `current_result` qui peuvent être appelées n'importe quand, tout comme `pause_restart` :

```
interruptible_algorithm :=
  !start_computation(INPUT_DATA) ;
    (  (  (?finished(RESULT) + ?error(ERROR_INFO))
            + (!stop_computation ; ?computation_stopped))
       ||  pause_restart
       ||  test_result) ;
       interruptible_algorithm

test_result :=
  (  (!get_current_result ; ?current_result(RESULT))
     ; test_result)
   + test_finished
```

3.4.4.5 Algorithme anytime

Un algorithme anytime étant un algorithme interruptible ayant un profil de qualité de la solution en fonction du temps, il suffit de reprendre l'une des instructions opératoires du paragraphe précédent et d'ajouter à l'artifact un attribut exhibant son profil de performance.

3.5 Méthodologie de modélisation d'une application avec des agents et des artifacts

Nous ne faisons qu'introduire un nouveau type d'entités dans les systèmes multi-agents. Et les méthodologies de conception de systèmes multi-agents déjà proposées comme Gaia [Wooldridge *et al.*, 2000] ou Voyelles [Demazeau, 1995] sont suffisamment générales pour prendre en compte les agents et l'environnement que nous décrivons ici.

Nous n'avons donc pas à redéfinir une nouvelle méthodologie complète. Nous nous contenterons simplement de bien mettre en évidence les différences entre les agents et les artifacts pour qu'il soit aisé de déterminer l'attribution des rôles et fonctionnalités du système aux agents et aux artifacts.

3.5.1 Différences entre agent et artifact

La principale caractéristique qui différencie les agents des artifacts est l'autonomie, au sens défini plus haut. Les agents sont autonomes alors que les artifacts ne le sont pas.

Un artifact, du fait qu'il ne soit pas du tout autonome, dispose d'un comportement prévisible de l'extérieur. Il peut donc exhiber un mode d'emploi qui permet d'interagir avec lui et de savoir à l'avance comment il va réagir. A l'inverse, un agent peut avoir un comportement imprévisible. On ne pourra donc pas disposer d'un mode d'emploi sûr. Lorsqu'un agent interagit avec un autre agent, il doit s'attendre à ce que son comportement réel ne soit pas celui énoncé par le modèle de fonctionnement éventuel dont il dispose. Les possibilités d'interaction avec un artifact sont différentes de celles possibles avec un agent, voire augmentées. Notons par exemple que l'autonomie des agents ne permet pas de les mettre en pause de l'extérieur, alors que cela devient possible avec les artifacts.

D'autres propriétés différencient les agents des artifacts. Elles découlent pour la plupart de la distinction autonome / pas autonome ci-dessus. C'est le cas de la pro-activité qui est une autre propriété généralement attribuée aux agents. Elle est la capacité à décider seul d'initier un comportement particulier (prise d'initiative). Ce nouveau comportement est décidé en interne, par l'expiration d'un timeout ou par un mécanisme de délibération, mais il ne peut être prévu de l'extérieur. Il n'y a donc que les entités autonomes qui peuvent être pro-actives. Un artifact ne peut pas dès lors être pro-actif, et ceci découle de la définition de l'autonomie que nous utilisons.

La réactivité est une autre propriété attribuée aux agents. Elle est utilisée dans deux sens différents. Un agent est dit réactif quand il a un comportement du type stimulus->réponse. C'est le comportement des objets de la POO. Les artifacts sont, dans ce sens, (hyper)réactifs : une action lancée est automatiquement exécutée. Les agents que nous concevons sont plutôt cognitifs que réactifs.

La réactivité d'un agent est également vue comme la capacité de prendre en compte les changements de l'environnement «à temps». L'introduction des artifacts de calcul a été faite initialement dans le but de permettre la réactivité des agents, en les déchargeant des calculs longs qui les empêcheraient de prendre en compte les informations venant de l'extérieur. Les artifacts sont plutôt introvertis et ne garantissent aucune réactivité par rapport aux changements dans l'environnement.

Un des autres points fondamentaux qui différencient les artifacts des agents est la manière dont on interagit avec eux. Suivant la tradition initiée par les travaux de John Searle, les agents interagissent entre eux par actes de langages : `tell`, `request`, ... Les agents ont le choix de prendre en compte ou non les messages qui leur parvienne.

Au contraire, un agent interagit avec un artifact au moyen des actions et des perceptions. Lorsqu'un agent effectue une action sur un artifact, celle-ci est automatiquement réalisée, sans que l'artifact ait le choix de refuser. La différence existe également au niveau des perceptions puisque les «réponses» faites par les artifacts sont prévues dans leur mode d'emploi. Aucun message ne peut arriver à un moment imprévu ou dépasser sa date limite d'arrivée.

Agent	**Artifact**
Autonome	Pas autonome
Proactif	Non proactif
Extraverti	Introverti
Dirigé par des buts	Orienté service

FIG. 3.7 – Principales différences entre les agents et les artifacts.

Notons enfin qu'il y a d'autres aspects pour lesquels les agents et les artifacts sont indifférenciés. C'est le cas pour l'aspect situé.

3.5.2 Classement des entités d'une application

Nous avons identifié un certain nombre de questions qu'il convient de se poser pour savoir si une entité de notre système doit être considérée comme un agent ou comme un artifact. Le tableau 3.7 est un aide mémoire des principales différences entre les agents et les artifacts. Il est expliqué dans les questions et réponses ci-dessous.

L'entité est-elle autonome ?

Comme nous l'avons remarqué dans le paragraphe précédent, une entité qui n'est pas du tout autonome sera un artifact. Une entité qui est autonome sera un agent. Les agents peuvent avoir différents degrés d'autonomie. Il sera parfois difficile de différencier un agent d'un artifact lorsque celui-ci ne dispose que de très peu d'autonomie. On se tournera alors vers les autres caractéristiques qui découlent de l'autonomie pour classer l'entité. Par exemple, la pro-activité est réservée aux agents. Les comportements des artifacts sont tous déclenchés par les actions demandées par les agents. Attention, cela n'empêche pas un artifact de déclencher lui-même des traitements internes quand bon lui semble. Il faut seulement que ceux-ci ne soient pas visibles de l'extérieur et que l'artifact puisse continuer à répondre immédiatement aux commandes qui lui sont envoyées.

L'entité est-elle extravertie ?

Si l'entité que nous avons identifiée a besoin d'être régulièrement à l'écoute du monde extérieur, on la concevra plutôt comme un agent. Si elle doit réaliser des tâches longues, elle seront déléguées à des artifacts de calcul pour lui permettre de rester extravertie. Au contraire, un artifact est une entité introvertie qui réalise entièrement la tâche qu'on lui a donnée avant de pouvoir dialoguer de nouveau. Cet aspect introverti apparaît nettement dans le mode d'emploi formel que l'on doit décrire pour tout artifact comme une succession d'actions, de réalisation d'action et d'envoi à l'agent d'une perception indiquant que la tâche est terminée.

Le comportement de l'entité est-il dirigé par des buts ?

Si c'est le cas, l'entité sera un agent. De plus, nous considérons qu'idéalement les agents doivent avoir une représentation explicite de leurs buts. Si cette représentation explicite des buts convient, il

est clair que l'entité sera un agent. Les artifacts seront au contraire orienté service. Ils n'ont aucun but explicite et se contentent de réaliser docilement les tâches qu'on leur ordonne de réaliser.

Puis-je définir un mode d'emploi précis que l'entité respectera ?

Lorsque le comportement d'une entité est très clairement défini et qu'il peut s'exprimer dans le langage de mode d'emploi fourni, il est préférable de considérer cette entité comme un artifact. Nous avions remarqué que concevoir des agents réellement autonomes pouvait poser beaucoup de problèmes dans l'implémentation des agents pour gérer toutes les réactions possibles des autres agents. L'introduction d'un maximum d'artifacts permet de limiter la complexité de programmation de chacune des entités. Attention cependant qu'il peut arriver que l'on arrive à définir un mode d'emploi précis pour une entité mais que celui-ci ne soit pas exprimable dans le langage de mode d'emploi disponible. On ne pourra dans ce cas pas considérer l'entité comme un artifact, mais comme un agent.

La fonctionnalité identifiée a-t-elle une persistance en dehors du cycle de vie des agents ?

La réponse à cette question permet d'avoir un argument supplémentaire pour choisir d'externaliser une fonctionnalité et ainsi créer un artifact pour la gérer.

La fonctionnalité que j'ai identifiée utilise-t-elle du code hérité sur lequel je n'ai pas ou peu de contrôle ?

Si c'est le cas, il faut l'instancier dans un artifact de calcul puisque ceux-ci ont été proposés pour fournir une encapsulation des codes hérités pour qu'ils soient aisément utilisables par les agents.

De plus, d'une manière générale, nous pensons qu'il convient d'essayer de ne conserver à l'intérieur des agents que ce qui est du niveau agent. Cela permet d'en simplifier la programmation et d'identifier plus clairement où il faut porter les efforts pour améliorer les performances des agents concernant la représentation des connaissances, la prise de décision, le suivi de contextes ou le focus d'attention rapide.

3.6 Comparaison avec les objets et les composants

Dans le but d'éclaircir encore plus la distinction entre les différents types d'entités d'un système, nous étendons notre comparaison aux objets et aux composants.

La distinction entre agent et objet a déjà clairement été définie notamment dans [Odell, 2002, Jennings *et al.*, 1998]. Nous la résumerons ainsi : *les objets font les choses gratuitement ; les agents le font pour de l'argent*. Lorsque qu'un objet A appelle une méthode sur un autre objet B, c'est l'objet A qui dispose du contrôle. Dans le cas agent, l'agent A doit *demander* à l'agent B de réaliser une action. Celui-ci peut refuser s'il ne voit aucun profit à réaliser l'action. Les artifacts se rapprochent en cela des objets puisque ce sont les agents qui en ont le contrôle. Les artifacts ne peuvent pas refuser de réaliser une action.

Les artifacts se placent cependant à un niveau d'abstraction différent de celui des objets. Les composants se placent au même niveau d'abstraction que les artifacts et leur structuration se rapproche très fortement de celle des artifacts. La figure 3.8 présente la structure d'un composant (ici un composant CORBA).

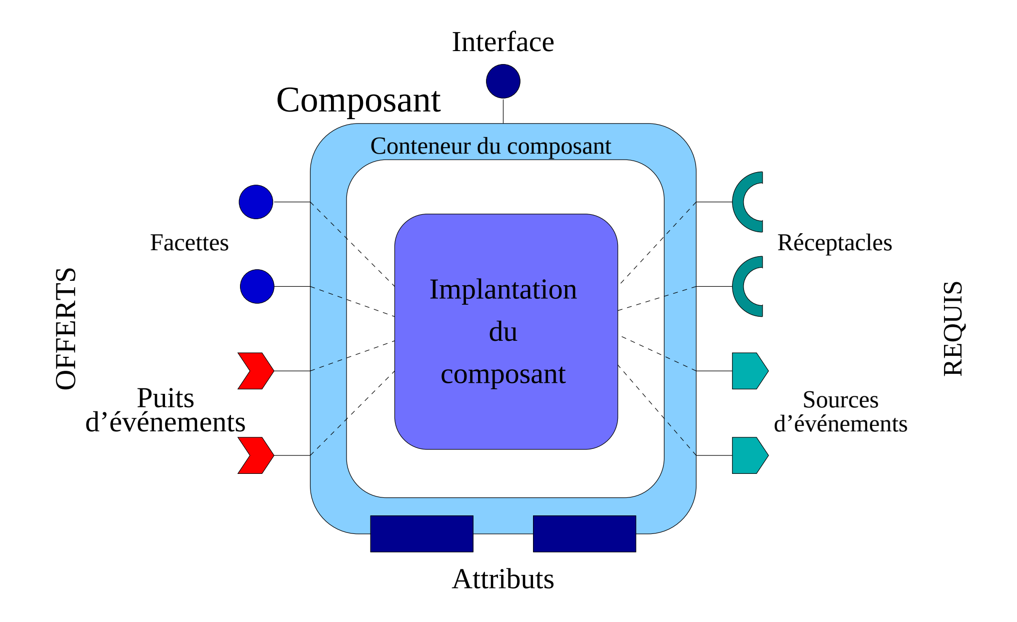


FIG. 3.8 – Caractéristiques d'un composant.

Un composant permet d'encapsuler des traitements. Il dispose d'une enveloppe (le conteneur) fournissant un certain nombre de services de base comme les possibilités de communication, de sécurité ou de persistance. On retrouve cette enveloppe dans les artifacts et les artifacts de calcul. Elle permet principalement les interactions avec les agents. Et, dans le cas des artifacts de calcul, elle gère directement certaines actions comme la mise en pause ou le redémarrage de l'artifact, qui sont des opérations de niveau système.

Les composants disposent de quatre types de ports :

– Une facette est une interface fournie par un type de composant et qui est utilisée par des clients en mode synchrone.
– Un réceptacle est une interface utilisée par un type de composant en mode synchrone.
– Un puits d'événement est une interface fournie par un type de composant et utilisée par ses clients en mode asynchrone.
– Une source d'événement est une interface utilisée par un type de composant en mode asynchrone.

Les artifacts quant à eux reçoivent les actions qu'ils doivent réaliser en mode synchrone : les actions sont réalisées dès qu'un agent le demande. Par contre, rien n'est imposé à l'agent pour le traitement des perceptions. Il peut le faire de manière synchrone ou asynchrone en gérant par exemple les perceptions de la même manière que les messages qu'il reçoit des autres agents. De plus, un

artifact n'utilise a priori pas les services d'autres entités du système, ou tout du moins ne les exhibe pas à l'extérieur. Il n'y a donc pas d'équivalent des réceptacles et des puits d'événements.

Les attributs représentent les propriétés configurables du composant. Ils se rapprochent des attributs définis pour les artifacts, mais leur sémantique est moins précise : dans les composants, ils pourraient ne pas exister et être remplacés par des méthodes appelées en mode synchrone pour lire ou modifier un attribut.

Ce qui différencie clairement un artifact d'un composant, ce sont les instructions opératoires qu'il exhibe. Les autres différences sont somme toute assez mineures. En définitive, et pour résumer, un artifact peut être considéré comme un composant disposant d'un mode d'emploi formel réifié que les agents peuvent manipuler. Cette proximité de concept doit certainement permettre d'accélérer le rapprochement dont nous parlions plus haut entre les communautés agents et composants.

3.7 Rapprochement avec les autres travaux de l'équipe SMAC

L'un des autres axes de recherche de l'équipe SMAC du LIFL concerne la modélisation centrée interaction. Dans cette approche, l'interaction entre deux entités est concrétisée et définie dans le SMA entre une source exécutant l'interaction et une cible la subissant.

Ce sont les interactions qui forment le cœur du modèle. Une interaction est une règle qui décrit formellement une action qu'un agent (acteur) peut déclencher sur un autre agent (cible).

SMAC propose dans [Mathieu et Picault, 2006, Mathieu et Picault, 2005] une méthodologie nommée IODA (Interaction-Oriented Design of Agent simulations) utilisant les concepts peut-subir/peut-effectuer pour la simulation multi-agents. La conception d'une simulation se fait en trois phases :

1. *détermination des interactions*. Cela revient à constituer un tableau indiquant pour chaque interaction la liste des agents qui pourraient la subir ou l'effectuer.
2. *détermination des agents sources et des agents cibles.* On donne pour chaque agent la liste des interactions qu'il peut subir et effectuer.
3. *détermination de la dynamique du système.* On dispose ici d'un moyen pour faire évoluer les possibilités d'interactions de chaque agent.

Les interactions ont une existence propre en dehors des agents qui les réalisent ou les subissent. Leur réification permet de les réutiliser dans différents contextes. Il en est de même avec les artifacts de calcul, qui réifient dans l'environnement du système multi-agent les outils de calcul utilisés par les agents pour atteindre leurs buts. On note une grande grande similitude avec nos artifacts de calcul. En effet, on peut considérer que, dans IODA, un agent qui ne peut rien effectuer est en quelque sorte un artifact. Il est complètement passif.

On retrouve dans ces travaux une classification en deux types d'entités que nous pouvons rapprocher de celle proposée dans cette thèse. Les agents animés sont à comparer aux agents décrits dans

cette thèse et les agents inanimés sont à comparer aux artifacts introduits précédemment. Nos agents sont des entités autonomes, cognitives et pro-actives. Ce sont les acteurs du système multi-agent, tout comme les agents animés de IODA. Les artifacts sont les objets du monde que peuvent utiliser les agents pour atteindre leurs buts. Il en est de même avec les objets inanimés de IODA.

Les agents inanimés peuvent subir des interactions, mais sont incapables d'en effectuer, tandis que les agents animés sont capables de subir et d'effectuer des interactions. Les agents inanimés sont les *objets* présents dans la simulation avec lesquels interagissent les agents animés, qui sont eux les *acteurs* de la simulation. Ne confondons pas ici animé avec mobile et inanimé avec statique. En effet, un objet animé peut très bien rester statique. C'est le cas par exemple d'un pommier qui produit des pommes. Par contre, un objet inanimé ne peut pas être mobile, à moins d'être déplacé par un agent animé. En effet, ne pouvant effectuer aucune interaction, il ne peut en particulier pas effectuer de déplacement.

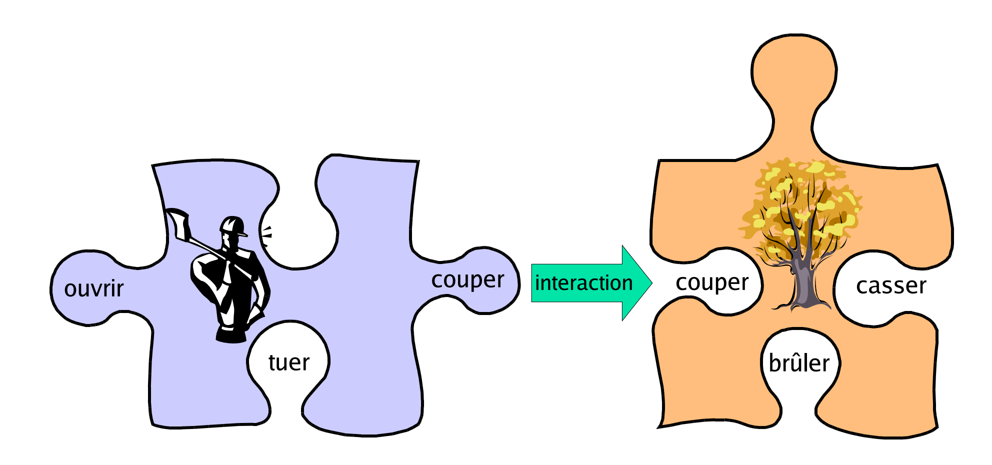


FIG. 3.9 – Peut-subir / peut-effectuer.

La figure 3.9 montre un exemple avec un agent animé `bûcheron` et un agent inanimé `arbre`. Le bûcheron peut ouvrir, couper et être tué. L'arbre peut être coupé, brûlé et cassé. La seule interaction possible en présence de ces deux agents est l'abattage de l'arbre par le bûcheron.

Les actions qu'un agent peut subir et effectuer sont exhibées à l'extérieur. Cela est à rapprocher de l'interface d'usage des artifacts qui indique les actions et perceptions disponibles au sein de l'artifact. Il n'y a cependant pas dans IODA d'équivalent des instructions opératoires car pour l'instant, les interactions entre les agents sont indépendantes les unes des autres et s'il fallait interagir de manière continue et ordonnée avec un autre agent, la suite des interactions serait déterminée par le moteur de planification de l'agent.

Remarquons cependant que dans le cadre de IODA, il n'a pas été nécessaire d'introduire la notion d'artifact. La notion la plus centrale est celle d'interaction et la séparation des agents en deux classes suffit. D'ailleurs, il n'y a pas du tout de différence entre effectuer une action si la cible est un agent animé ou un agent inanimé. Par exemple, un agent peut indifféremment effectuer l'interaction `taper`

sur une porte ou sur un autre agent ! Avec les agents et les artifacts, il y a une différence, soulignée plus haut, entre les interactions avec d'autres agents ou avec un artifact. Dans le premier cas, on interagit au moyen d'actes de langages ; et dans le second cas, au travers des actions et des perceptions.

Cette approche centrée interaction est utilisée chez SMAC à la fois dans des systèmes cognitifs (projet CoCoA) et dans des systèmes réactifs (projet SimuLE).

CoCoA a pour objectif d'appliquer cette approche aux systèmes cognitifs. Dans CoCoA, l'interaction est composée de trois parties [Devigne *et al.*, 2005a, Devigne *et al.*, 2005b, Devigne *et al.*, 2004] :

– *les conditions.* Ce sont les conditions nécessaires à l'exécution de l'interaction. Elles portent sur des propriétés des agents de la simulation. L'agent qui veut déclencher une interaction doit d'abord résoudre ces conditions.
– *la garde.* C'est une condition particulière qui a été isolée des autres pour gérer l'aspect situé des simulations. Elle spécifie la distance à la cible qu'il faut respecter avant de pouvoir réaliser une interaction.
– *les actions.* Ce sont les conséquences de l'exécution de l'interaction. Elles portent sur les propriétés des agents ou les agents eux-mêmes (création, destruction, ...).

[Devigne *et al.*, 2005c] s'attaque principalement au problème de la gestion d'équipes pour lesquelles un chef distribue des tâches aux autres membres de l'équipe. L'autonomie des membres de l'équipe s'exprime par le fait que le chef ne connaît pas les actions atomiques pour atteindre un but. Il possède seulement la connaissance de tâches abstraites. Il délègue aux autres agents la réalisation de chacun des sous-buts qu'il aura déterminés. Les agents pourront ainsi déterminer eux-même la meilleure stratégie pour mener à bien leur mission. Ils disposent tous pour cela d'un moteur de planification en chaînage arrière. A tout moment, le chef peut décider de réaffecter une nouvelle tâche à un agent[11], celle-ci remplaçant le but précédent. Leur autonomie est donc une autonomie de décision, dans le cadre donné par le chef de l'équipe.

SimuLE (Simulation Large Échelle) applique le modèle de IODA aux systèmes réactifs et s'attache particulièrement à montrer son intérêt dans les simulations à très grand nombre d'individus. Le domaine d'application principalement ciblé est celui de la biologie cellulaire.

3.8 Synthèse

Nous avons identifié le besoin d'isoler et de traiter de manière particulière les tâches de calcul de nos agents. Nous nous sommes donc posés la question de la nature des entités qui réalisent ces calculs.

Nous nous sommes aperçus que ces traitements avaient intérêt à être externalisés pour notamment ne pas trop complexifier l'architecture des agents. Les tâches externalisées pourraient être attribuées

[11]Notons que les replanifications faites par le chef tentent de minimiser le nombre de réaffectations nécessaires. Cette réaffectation intelligente se fait de manière incrémentale, tout en garantissant une solution qui minimise le nombre de réaffectations, et constitue une partie des travaux de thèse de Damien Devigne.

à une classe particulière d'agents, mais cela ne convient pas, notamment à cause de la propriété d'autonomie des agents, à laquelle nous donnons une définition qui convient à notre contexte d'étude.

J'ai proposé de réutiliser la notion d'artifact introduite par Ricci, Viroli et Omicini en 2004. Celle-ci nous permet d'introduire dans nos systèmes des artifacts de calcul qui prennent en charge les tâches calculatoires longues des agents. Les artifacts disposent d'instructions opératoires qui décrivent leur mode d'emploi précis et garanti que les agents doivent suivre au mieux.

Nous avons proposé quelques extensions au modèle de base permettant de mieux prendre en compte les contraintes temporelles liées à nos artifacts de calcul et également pour lui apporter plus de flexibilité. Cela nous permet d'avoir un mécanisme de récupération après une erreur et d'utiliser des informations sur l'historique des états des instructions opératoires pour gérer des boucles simples dans le modèle proposé par l'équipe italienne.

Les possibilités d'extension du modèle actuel ne seront pas suffisantes pour l'avenir. Nous avons notamment identifié quelques fonctionnalités intéressantes pour lesquelles une refonte du modèle de description des modes d'emploi des artifacts est nécessaire.

Enfin, nous avons fait un point méthodologique qui décrit comment les agents et les artifacts se différencient de manière conceptuelle. Nous avons aussi décrit comment, dans une application donnée, on peut distinguer quelles entités seront des agents et quelles autres seront des artifacts. Nous avons également fait une comparaison avec les objets, les composants et d'autres travaux de l'équipe SMAC sur les systèmes multi-agents centrés interaction.

Chapitre 4

La gestion des ressources processeur

Il faut toujours plus de temps que prévu, même en tenant compte de la loi de Hofstadter.
Loi récursive de Hofstadter.

Le présent n'est pas un passé en puissance, il est le moment du choix et de l'action.
Simone de Beauvoir.

L'avenir est ce qu'il y a de pire dans le présent.
Gustave Flaubert.

Nous nous intéressons dans ce chapitre à la manière dont nous devons gérer le partage des ressources processeur pour que nos agents puissent disposer d'un comportement extraverti et que l'on puisse donner des dates limites aux tâches effectuées par les artifacts de calcul. Nous avons déjà remarqué dans les sections 2.1.3.1 et 2.2.2.1.2 que les systèmes temps réel ne sont d'aucune utilité dans notre cadre. En effet, ils imposent des contraintes très fortes au niveau de la programmation des processus pour lesquels on veut assurer des délais d'exécution. Nous préférons de plus que nos SMA puissent fonctionner sur les systèmes d'exploitation classiquement installés sur les machines de bureau comme Windows ou Unix, qui ne sont pas temps réel. Nous pouvons ainsi utiliser les propriétés liées à la fonctionnalité de temps partagé fournie par ces systèmes.

Nous considérons tout d'abord que nous disposons d'un unique processeur sur lequel s'exécutent les agents et les artifacts. Nous verrons ensuite comment il est possible d'étendre le modèle que nous proposons à de multiples processeurs.

Nous attendons des agents qu'ils s'intéressent à leur environnement. Ils doivent être en permanence à l'écoute de l'extérieur pour prendre rapidement en compte les changements dans l'environnement et répondre aux sollicitations des autres agents. Pour des agents logiciels, cela revient à disposer

régulièrement d'assez de temps processeur qu'ils utiliseront pour se remettre en phase avec leur environnement.

Nous définissons les périodes d'activité et d'inactivité des entités de nos systèmes et nous associons aux artifacts une horloge « temps artifact » qui permet d'évaluer la quantité de travail qu'un artifact a réalisé à un instant donné.

Définition 4.1 (Entité active/ entité inactive) *Une entité du SMA est inactive sur une période de temps donnée si elle ne dispose d'aucun accès processeur sur cette période. Elle est active sur la période considérée dans le cas contraire.*

Définition 4.2 (Temps artifact) *Le temps artifact avance au rythme des opérations exécutées par l'artifact.*

Le temps artifact est propre à chaque artifact : il y a une horloge « temps artifact » par artifact qui peut avancer à un rythme différent pour chacun d'eux.

Nous avons présenté dans le chapitre précédent comment il est possible de décrire le mode d'emploi de tous types d'algorithmes encapsulés dans les artifacts. En particulier, les agents sont capables de déléguer des calculs dont les durées (en temps artifact) ne sont pas forcément connues avant de les avoir réalisés. C'est le cas des algorithmes que nous avons appelé « boîtes noires ». Un agent qui veut avoir la maîtrise de ses calculs « boîtes noires » ne va pas se lancer dans l'exécution complète et en une seule fois de ceux-ci. Il va plutôt décider de les commencer pendant un certain temps, de voir où il en est arrivé et d'éventuellement les continuer pendant une nouvelle période de temps. Il fera cela jusqu'à la terminaison de ses traitements ou jusqu'à ce qu'il renonce à les terminer. C'est pourquoi nous faisons dans la suite la distinction suivante entre les mots *traitement* et *tâche*.

Définition 4.3 (Traitement / tâche) *Un traitement est une instanciation d'un algorithme sur des données d'entrée particulières. Une tâche est l'exécution d'un traitement pendant un temps artifact fixé.*

Un traitement peut être réalisé en une ou plusieurs tâches. Lorsqu'une tâche se termine, une nouvelle peut continuer le traitement où il en était arrivé. Si l'on fait le lien avec le langage de mode d'emploi des artifacts proposé dans le chapitre précédent, un traitement est la réalisation d'une instance de mode d'emploi d'un artifact jusqu'au bout de celui-ci. Les modes d'emploi que nous fournissons pour nos artifacts de calcul spécifient que l'agent dispose à tout moment de la possibilité d'appeler les actions `pause` et `resume` ou `pause_computation` et `resume_computation`. Une tâche sera donc définie sur la période de temps entre deux mises en pause.

4.1 Gestion des ressources processeur avec des délais à respecter

Nous considérons dans cette section que les agents découpent les traitements aux durées indéterminées qu'ils délèguent aux artifacts en tâches (voir définition 4.3). À notre niveau, nous connaissons donc pour chaque tâche réalisée par les artifacts la durée exacte en temps artifact qui sera nécessaire. Nous considérons également qu'à chaque tâche est associée une date limite, donnée en temps APRC, à laquelle elle doit être terminée. Les dates limites concernent uniquement les calculs réalisés par les artifacts : les calculs réalisés en interne par les agents n'ont aucune contrainte temporelle à respecter.

4.1.1 Besoin d'une entité gérant les ressources processeur

Les SMA tels que nous les avons décrits dans le chapitre précédent sont uniquement constitués d'agent et d'artifacts de calcul. Chaque agent étant autonome, il peut prendre à tout moment la décision de faire effectuer une tâche de calcul à un de ses artifacts. Si les agents ne disposent pas de moyens pour coordonner leur utilisation du processeur, ils n'arriveront pas à respecter leurs délais sur les tâches de calcul.

Le seul moyen d'assurer que les agents pourront tenir leurs délais est de calculer un ordonnancement pour l'ensemble des tâches exécutées sur le processeur.

Le calcul de l'ordonnancement pourrait être décentralisé au sein de chaque agent. Lorsqu'un agent aurait une nouvelle tâche à réaliser, il calculerait le nouvel ordonnancement et indiquerait aux autres agents les informations sur la planification de la nouvelle tâche. Cette solution, bien que tout à fait dans l'esprit agent, est pourtant totalement inefficace et très difficile à mette en œuvre. En effet, il faudrait diffuser des informations à tous les agents à chaque nouvelle tâche planifiée. Il serait également très difficile d'assurer que tous les agents disposent bien des toutes dernières informations avant de lancer le calcul de l'ordonnancement pour une nouvelle tâche.

Comme nous nous plaçons dans un premier temps sur un unique processeur, nous considérons que la meilleure solution est d'intégrer le calcul de l'ordonnancement au sein d'une entité dédiée et unique.

4.1.2 Un artifact de coordination pour le partage du processeur

Nous proposons d'utiliser un artifact de coordination pour gérer l'accès aux ressources du processeur. Il sera nommé Artifact de Partage des Ressources de Calcul (APRC). Son rôle sera de calculer l'ordonnancement pour l'ensemble des processus s'exécutant au sein du SMA : les agents, les artifacts de calcul et l'APRC lui-même. Il ne devra pas nuire à l'autonomie des agents. C'est pourquoi, lorsqu'une nouvelle tâche ne pourra être planifiée à cause d'une surcharge du système, l'APRC ne prendra aucune décision relative aux tâches déjà planifiées pour tenter d'allouer du temps processeur

à la nouvelle tâche. Il laissera cette décision aux agents eux-même. Il jouera le rôle d'intermédiaire entre les agents durant cette phase de négociation.

Nous avons vu précédemment que chaque artifact dispose d'une horloge personnelle qui lui indique son «temps artifact». Les agents quant à eux doivent se référer dans leurs raisonnements à une horloge, quelconque, pour être conscients du temps. L'APRC va devoir collecter les dates limites des tâches des différents artifacts. Ces dates doivent toutes se référer à une horloge commune. Nous définissons à cet effet le «temps APRC» :

Définition 4.4 (Temps APRC) *Le temps APRC est un temps commun à tous les agents qui utilisent les services de l'APRC.*

On choisira souvent l'horloge système pour donner le temps APRC, mais il est possible que ce soit une horloge virtuelle qui le donne, par exemple dans le cas de simulations pour pouvoir compresser ou dilater le temps. Le «temps artifact» peut être utilisé pour des raisonnements qui doivent être indépendants du processeur et de sa charge. Au contraire, le «temps APRC» est utilisé par des raisonnements qui dépendent de la charge actuelle du processeur.

4.1.3 Algorithme d'ordonnancement

4.1.3.1 Propriétés nécessaires

L'ordonnanceur dont nous avons besoin doit disposer de propriétés particulières. En se plaçant au-dessus de celui du système d'exploitation, il peut prendre en compte le fait qu'à son niveau, plusieurs tâches peuvent s'exécuter en parallèle. Notamment, il doit prendre en compte de manière particulière les agents qui doivent toujours être considérés comme actifs pour conserver leur propriété d'extraversion. Il doit pouvoir gérer des dates limites, tout en étant facilement extensible à d'autres contraintes temporelles.

Contrairement à ce que font la plupart des ordonnanceurs utilisés pour gérer des dates limites, nous considérons qu'il est intéressant de commencer les tâches au plus tôt. En effet, si nous considérons des algorithmes incrémentaux ou dont le temps d'exécution est très variable, il est intéressant pour l'agent de commencer à exécuter ses tâches de calcul au plus tôt et ainsi avoir le plus tôt possible des informations quant au comportement de l'algorithme (convergence lente ou rapide par exemple).

Le comportement d'un ordonnanceur comme EDF [Liu et Layland, 1973], qui va exécuter entièrement la tâche dont la date limite est la plus proche puis entièrement la tâche suivante, n'est notamment pas adapté à la résolution d'un même problème par différents artifacts avec différentes heuristiques. Dans ce genre d'application, il est intéressant de démarrer au plus tôt les différentes heuristiques et d'arrêter la résolution dès qu'un artifact a trouvé la solution recherchée.

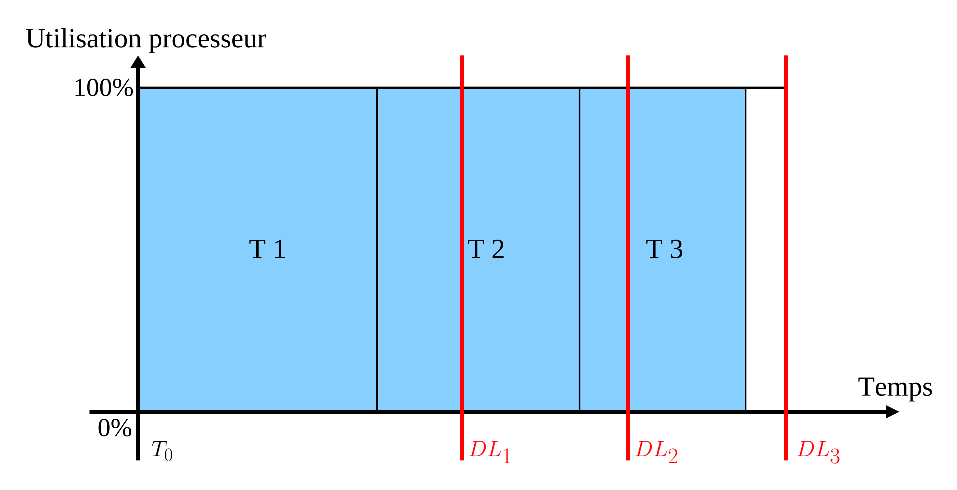


FIG. 4.1 – Exemple d'ordonnancement obtenu avec Earliest Deadline First.

La figure 4.1 donne un exemple d'ordonnancement obtenu avec EDF pour trois tâches T1, T2 et T3 dont les dates limites respectives sont DL_1, DL_2 et DL_3.

L'ordonnancement réalisé par l'APRC doit prendre en compte l'utilisation qu'il fait lui-même du processeur. Il en utilisera en effet une part non négligeable pour réaliser l'ordonnancement, pour dialoguer avec les agents se trouvant sur le même processeur que lui et également pour dialoguer avec les autres APRC se trouvant sur d'autres processeurs (voir la section 4.3 pour ce dernier point).

4.1.3.2 Un algorithme qui répond à nos besoins

Nous présentons maintenant l'algorithme que nous avons mis au point pour répondre à notre besoin d'ordonnancement. Le but est ici de prouver la faisabilité d'un système prenant en compte les contraintes précédemment citées. Nous détaillons plus loin les avantages et les inconvénients de notre solution.

4.1.3.2.1 Modélisation

Les requêtes sont sous la forme $[ET_i,\ DL_i]$. ET_i est le temps artifact à affecter à la tâche i avant la date limite DL_i. On parlera également pour ET_i d'énergie de calcul totale à attribuer à la tâche i. On considère n requêtes dont les dates limites sont triées par ordre croissant : $DL_{i+1} \geq DL_i, \quad 1 \leq i \leq n-1$. On obtient n intervalles : $[T_0$ à DL_1, DL_1 à DL_2, ...,DL_{n-1} à $DL_n]$, avec T_0 la date de début de l'ordonnancement. On considère également un nombre nb_agents fixe d'agents actifs en permanence.

Nous voulons respecter les dates limites associées à chaque tâche : pour le $i^{ème}$ intervalle, le but est donc de terminer la $i^{ème}$ tâche avant la fin de l'intervalle. Nous voulons aussi donner le plus possible de temps processeur aux tâches $i+1$ à n et ceci le plus tôt possible dans l'intervalle. On partitionne l'intervalle considéré en sous-intervalles correspondants à toutes les possibilités de faire tourner la tâche i en parallèle d'une ou plusieurs autres des tâches $i+1$ à n. Pour le $i^{ème}$ intervalle, il y a 2^{n-i} sous-intervalles. On ordonne les sous-intervalles par ordre décroissant du nombre de tâches en faisant partie. Pour un sous-intervalle comportant T tâches, chaque tâche dispose d'une part $P_{i,j}$ de la puissance processeur disponible égale à $\frac{1}{1+T+nb_agents}$, le 1 étant présent pour prendre en compte l'utilisation processeur de l'APRC.

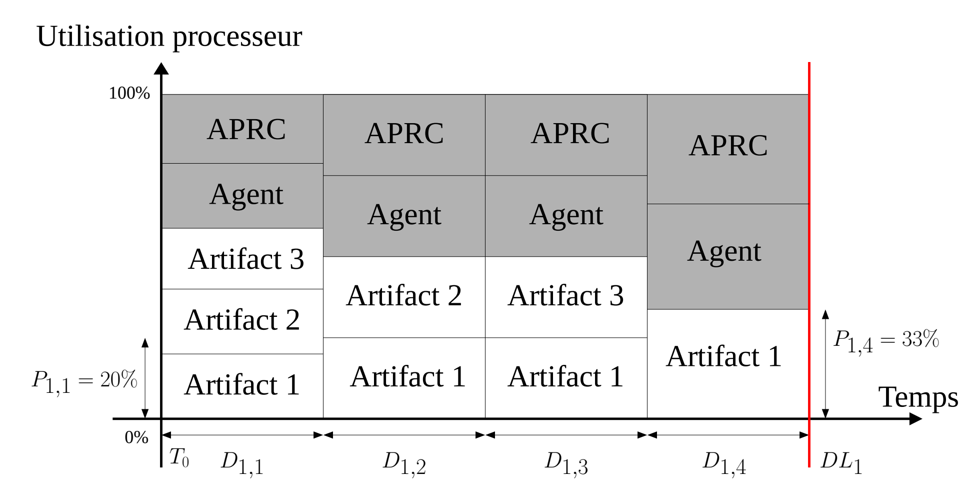


FIG. 4.2 – Partitionnement du premier intervalle dans le cas d'1 agent, de 3 artifacts et de l'APRC.

Ce partitionnement d'un intervalle i correspond bien à ce que nous voulons faire : la priorité est donnée à la terminaison de la tâche i puisqu'elle apparaît dans tous les sous-intervalles et si on peut exécuter une ou plusieurs autres tâches en parallèle de la tâche i, on le fera le plus tôt possible dans l'intervalle. Le problème pour le $i^{ème}$ intervalle revient donc à déterminer les durées $D_{i,j}$, $D_{i,j}$ étant la durée du sous-intervalle j de l'intervalle i. Les $D_{i,j}$ peuvent s'annuler.

4.1.3.2.2 *Préparation de la résolution*

La résolution est faite intervalle par intervalle. On calcule pour chaque intervalle les variables nécessaires à la constitution d'un système d'équations et d'inéquations linéaires que l'on résout grâce à l'algorithme du simplexe [Faure, 1979].

On définit tout d'abord les limites de l'intervalle considéré :
- la date de fin est toujours égale à DL_i,
- la date effective de début DD_i peut être différente de DL_{i-1} si la tâche $i-1$ s'est terminée avant la fin de son intervalle. Elle est obtenue par :

$$DD_1 = T_0$$

et

$$DD_i = \sum_{j=1}^{2^{n-i-1}} D_{i-1,j}, \quad 1 \leq j \leq 2^{n-1}$$

Soit $L_{i,j,k}$ qui vaut 1 si la tâche k possède une de ses parties dans le sous-intervalle j de l'intervalle i et 0 sinon. Et soit $E_{i,k}$ l'énergie totale effectivement affectée à la tâche k dans l'intervalle i.

$$E_{i,k} = \sum_{j=1}^{2^{n-k}} L_{i,j,k} P_{i,j} D_{i,j}$$

On peut calculer $ER_{i,k}$, l'énergie restante pour la tâche k à partir de l'intervalle i inclus, ainsi :

$$ER_{1,k} = ET_k$$

et

$$ER_{i,k} = ER_{i-1,k} - E_{i-1,k}, \quad 2 \leq i \leq n, \quad 1 \leq k \leq 2^{n-1}$$

Comme nous résolvons le problème intervalle par intervalle, nous devons nous assurer que nous choisissons correctement les tâches qui vont s'exécuter dans un intervalle i particulier. En effet, pour les tâches qui ne pourraient pas s'exécuter entièrement dans leur intervalle, nous devons prendre en compte dans les intervalles précédents que l'on doit absolument leur affecter une part de l'énergie disponible. Un exemple dans le paragraphe 4.1.3.2.4 illustre cela.

Soit ESR_i l'énergie minimale à affecter dans l'intervalle i à d'autres tâches que la $i^{ème}$. Et soit $C_{i,k}, \quad i+1 \leq k \leq 2^{n-1}$ qui vaut 1 si la tâche k fait partie de la liste des tâches pour lesquelles il faut affecter un minimum d'énergie dans l'intervalle i. Le calcul des $C_{i,k}$ et de ESR_i se fait en partant du dernier intervalle et en remontant vers l'intervalle i. On initialise tout d'abord ESR_i et les $C_{i,k}$ à 0. A l'intervalle m, on met à jour ESR_i et les $C_{i,k}$ de la manière suivante :

– Soit M_m l'énergie processeur disponible dans l'intervalle m.
  $M_m = P(DL_m - DD_m)$ avec P la puissance processeur par unité de temps.
– Si $ER_{i,m} + ESR_i \leq M_m$, alors on réinitialise ESR_i et les $C_{i,k}$ à 0.
– Sinon, on ajoute $(ER_{i,m} - M_m)$ à ESR_i et on place $C_{i,m}$ à la valeur 1.

4.1.3.2.3 Résolution

Nous avons maintenant déterminé toutes les valeurs numériques utiles et il ne reste plus que les $D_{i,j}$ comme variables à déterminer. Le programme linéaire à résoudre pour cela est le suivant :

$$Max\ z = \sum_{j=1}^{2^{n-i}} [(2^{n-i} - j + 1) D_{i,j}] \tag{4.1}$$

$$D_{i,j} \geq 0, \quad 1 \leq j \leq 2^{n-i} \tag{4.2}$$

$$\sum_{j=1}^{2^{n-i}} D_{i,j} \leq DL_i - DD_i \tag{4.3}$$

$$E_{i,i} = ER_{i,i} \tag{4.4}$$

$$E_{i,k} \leq ER_{i,k}, \quad j+1 \leq k \leq 2^{n-1} \tag{4.5}$$

$$\sum_{k=j+1}^{2^{n-1}} (C_{i,k}E_{i,k}) \geq ESR_i \tag{4.6}$$

(4.1) On cherche à maximiser la somme pondérée des durées $D_{i,j}$ de l'intervalle i. Les facteurs de pondération sont tels que le facteur de $D_{i,j}$ soit plus grand que le facteur de $D_{i,j+1}$.

(4.2) Toutes les durées sont positives.

(4.3) La tâche i doit se terminer avant sa date limite : la somme des durées de ses parties est inférieure ou égale à la durée de l'intervalle courant.

(4.4) La somme des énergies des parties de la tâche i dans l'intervalle i doit être égale à l'énergie restante à affecter pour la tâche i.

(4.5) On ne peut affecter plus d'énergie à une tâche dans l'intervalle i que la quantité qui reste pour cette tâche à partir de cet intervalle.

(4.6) Il faut absolument consacrer la quantité ESR_i de l'énergie disponible dans l'intervalle courant pour les tâches k dont la variable $C_{i,k}$ vaut 1.

4.1.3.2.4 Exemples de résolution

Nous considérons pour ce paragraphe que le processeur peut exécuter 1000 opérations par seconde. Les requêtes de tâches $[ET, DL]$ sont formulées ainsi :
- ET est une durée en temps artifact donnée en nombre d'instructions à réaliser,
- DL est la date limite pour la tâche en temps APRC donnée en secondes.

Pour simplifier notre premier exemple, nous occultons le fait que l'APRC et qu'un agent s'exécutent. Nous considérons uniquement trois artifacts A1, A2 et A3 qui exécutent chacun une tâche de calcul. Les requêtes sont : $[3000ops, 4s]$, $[2700ops, 6s]$ et $[2000ops, 8s]$. Nous remarquons qu'il est impossible d'exécuter entièrement la tâche 2 uniquement dans le second intervalle. Celui-ci a une durée de deux secondes et donc une capacité d'exécution de seulement 2000 opérations. Il faut exécuter au moins 700 opérations de la tâche 2 dans le premier intervalle. La figure 4.3 illustre cela : on doit arrêter la tâche 3 qui tournait en parallèle de 1 et 2 pour que la somme des parties de la tâche 2 dans le premier intervalle puisse atteindre la valeur de 700 opérations.

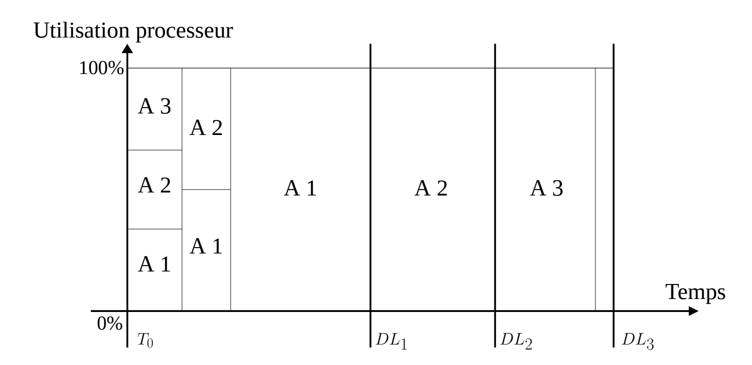


FIG. 4.3 – Prise en compte dans la résolution pour un intervalle de contraintes sur les intervalles restants.

Prenons, pour notre second exemple, un agent, l'APRC et les requêtes suivantes pour trois artifacts : $[700ops, 4s]$, $[1000ops, 6s]$ et $[1300ops, 8s]$. La figure 4.4 montre que l'on arrive à terminer la tâche 1 avant la fin de son intervalle tout en exécutant les tâches 2 et 3 en parallèle. La nouvelle date de début du second intervalle n'est plus DL_1, mais la somme des $D_{1,j}$. Il en est de même pour la date de début du troisième intervalle puisque la tâche 2 se termine avant la date DL_2.

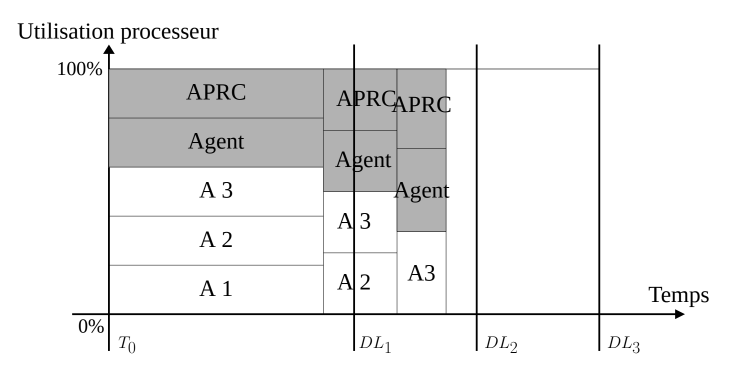


FIG. 4.4 – Modification des dates de début d'intervalles.

4.1.4 Extensions et variantes de l'algorithme d'ordonnancement

La description précédente de notre algorithme d'ordonnancement prend en compte un cas d'utilisation assez strict. Nous proposons ici différentes extensions et variantes qui peuvent permettre de s'adapter à différentes situations que l'on peut rencontrer en pratique lorsqu'on utilise notre ordonnanceur.

4.1.4.1 Apparition de nouveaux agents

L'algorithme présenté plus haut gère un nombre d'agents fixe et un nombre d'artifacts évolutif. Il est possible d'étendre le système pour prendre en compte un nombre d'agents variable. Il faut cependant prendre garde à ce que les nouveaux agents ne démarrent pas n'importe quand car cela empêcherait les artifacts de tenir leurs délais.

Il faut donc l'autorisation de l'APRC pour que les nouveaux agents démarrent à un moment où ils ne perturbent pas les engagements pris auparavant. Cela apporte juste quelques modifications dans la primitive qui permet de lancer l'exécution d'un nouvel agent. Cette primitive ne lance pas tout de suite le nouveau processus mais envoie une requête à l'APRC pour savoir à quelle date planifier la création du nouvel agent.

Dans l'implémentation actuelle, l'APRC renvoie systématiquement en réponse à cette requête la date limite la plus lointaine pour l'ensemble des engagements actuels. Cela permet de ne pas devoir recalculer un nouvel ordonnancement pour l'ensemble des tâches déjà planifiées. L'APRC prend en compte qu'à partir de la date qu'il a renvoyée, il y aura un agent de plus sur le processeur et donc les futures tâches seront planifiées en prenant en compte cette information.

4.1.4.2 Utilisation processeur des agents

Nous avons également étudié la portée de l'hypothèse faite sur l'utilisation processeur des agents. Nous considérons dans l'algorithme d'ordonnancement qu'un agent dispose de $\frac{1}{1+nb_agents+nb_artifacts}$ du processeur.

Quand il y a une proportion plus importante d'agents que d'artifacts, la part de ressources réservée à ces derniers devient très faible. Cette hypothèse est valable si l'on considère que les agents ont toujours des traitements à réaliser. Cependant, en pratique on remarque que les agents se limitent à des automates de traitement des messages entrants. Le traitement d'un message entrant ne nécessite souvent que très peu de ressources puisque les traitements longs sont délégués aux artifacts. Les agents sont donc souvent en pause, en attente d'un message à traiter. Ce phénomène est accentué dès lors que les agents sont synchronisés par l'utilisation de protocoles (un agent envoie un message à un autre agent et attend sa réponse pour continuer). La synchronisation uniformise la quantité de ressource totale réellement utilisée par les agents : il n'y aura pas de moments où tous les agents vont s'exécuter en même temps.

Dans ces conditions, il serait plus judicieux d'utiliser l'hypothèse que l'ensemble des agents n'utilise pas plus qu'une certaine fraction des capacités du processeur. Il est juste nécessaire que cette hypothèse soit vraie en moyenne sur chaque intervalle séparant deux dates limites. D'autres hypothèses peuvent être imaginées, en fonction du SMA en construction.

Le choix d'une hypothèse particulière se fait très facilement dans un SMA donné en changeant l'attribution des $P_{i,j}$ dans l'algorithme d'ordonnancement.

4.1.5 Exécution de l'ordonnancement

L'ordonnancement est calculé par l'APRC. Il doit ensuite être exécuté, c'est-à-dire qu'une des entités du SMA doit se charger d'envoyer les commandes de pause et de redémarrage aux artifacts aux limites des intervalles de temps déterminés par l'APRC. Nous proposons trois modes de fonctionnement : gestion par l'artifact lui-même, gestion par l'APRC et gestion par l'agent qui délègue sa tâche à l'artifact concerné.

Nous détaillons dans les paragraphes suivants ces trois modes de fonctionnement et donnons pour chacun d'eux les avantages et inconvénients.

4.1.5.1 Gestion par l'APRC

On peut très facilement charger l'APRC de démarrer et arrêter les artifacts au bon moment, puisqu'il dispose de toutes les informations sur l'ordonnancement du système. Il suffit qu'il fasse appel au système d'exploitation pour être réveillé au moment d'effectuer une mise en pause ou un redémarrage d'un artifact. Cela peut se faire au moyen des signaux système ou des mécanismes d'alarme fournis par le système d'exploitation. La mise en pause et le réveil des artifacts se fera comme décrit dans la section 3.3.3 : soit par envoi de signaux systèmes SIGSTOP et SIGCONT, soit par envoi de messages sur une socket constamment surveillée par l'artifact.

L'encapsulation de ce travail dans l'APRC est correcte du point de vue modélisation de l'application et elle permet de limiter au maximum les appels aux mécanismes de réveil du système d'exploitation : un réveil du système d'exploitation correspond à une limite d'un sous-intervalle de l'ordonnancement et donc potentiellement à plusieurs artifacts à activer ou désactiver. Les systèmes d'exploitation de permettent cependant de programmer qu'une alarme par processus. Il faut donc gérer de manière explicite au niveau de l'APRC la liste des alarmes pour l'ensemble des artifacts et transférer au système d'exploitation à chaque réveil la date du prochain réveil.

4.1.5.2 Gestion par l'artifact lui-même

Si l'on veut que ce soit l'artifact qui se charge lui-même de se mettre en pause et de redémarrer tout seul, on peut utiliser un mécanisme similaire à celui précédemment décrit mais dupliqué dans chacun des artifacts. Cette solution est moins facile à mettre en place car il faut ajouter dans chaque artifact une couche chargée de demander au système d'exploitation de lui envoyer un signal d'attention à chaque limite d'intervalle. Elle est également moins efficace que la solution où tout est géré par l'APRC car chacun des artifacts qui doivent se réveiller ou s'endormir à une date donnée vont planifier un réveil au niveau du système d'exploitation. La gestion des alarmes multiples est en grande partie déléguée au système d'exploitation et il devient possible d'utiliser d'autres types d'alarmes que les alarmes temps réel du système d'exploitation. On pourrait par exemple utiliser des alarmes en temps CPU pour assurer que les artifacts s'exécutent pendant un temps CPU donné.

4.1.5.3 Gestion par l'agent

On peut enfin vouloir déléguer l'exécution de l'ordonnancement d'un artifact à son agent propriétaire. On retrouve ainsi l'avantage que nous avions lorsque c'était l'APRC qui en était chargé : on peut gérer la pause et le redémarrage entièrement de l'extérieur sans avoir à modifier le code des artifacts.

Cette solution peut permettre de flexibiliser l'attribution des tranches de temps aux artifacts d'un agent. En effet, un agent pourrait estimer à un moment qu'un de ses artifacts (A1) aurait un besoin en ressources processeur un peu plus important que prévu et qu'il pourrait sacrifier pour cela une tranche de temps allouée à un de ses autres artifacts (A2). Dans ce cas, avec le mode de fonctionnement normal, il devrait indiquer à l'APRC qu'il renonce à une partie des tranches temporelles allouées à A2 et faire une requête pour allouer de nouvelles tranches de temps pour A1. Si, au contraire, c'est l'agent qui gère l'exécution de l'ordonnancement, il pourrait prendre la décision de lancer A1 plutôt qu'A2 dans certaines tranches de temps et ceci sans en informer l'APRC.

Cette solution peut également permettre aux agents de raisonner sur les tranches de temps qui sont allouées à leurs artifacts. Pour cela, on peut exprimer l'ordonnancement sous la forme d'une instruction opératoire que l'on vient insérer sous la forme d'une branche parallèle dans le mode d'emploi de l'artifact.

Cette solution a cependant un inconvénient majeur. En effet, à chaque ajout d'une tâche dans le planning du processeur, les intervalles de temps dans lesquels les artifacts doivent être lancés peuvent être modifiés pour l'ensemble des tâches précédemment validées. Cela implique que si un agent veut raisonner sur les intervalles de temps alloués à ses artifacts, il doit remettre à jour ses raisonnements à chaque fois qu'une nouvelle tâche est allouée et qu'un nouveau tissage a été effectué sur les modes d'emploi de ses artifacts.

4.1.6 Mode d'emploi de base de l'APRC

Nous présentons ici la partie du mode d'emploi de l'APRC qui permet à un agent de faire une requête pour une nouvelle tâche. Si la tâche peut être réalisée, l'APRC renvoie les intervalles de temps dans lesquels la tâche peut s'exécuter. Si la tâche ne peut être allouée, la réponse est simplement un message de refus.

`Def_CA_Management` est l'instruction opératoire utilisée par les agents pour interagir avec l'APRC lorsqu'ils ont un travail à donner à un artifact de calcul. L'agent exécute tout d'abord l'action `request(r_content)`. Nous considérons que `r_content` est un terme à quatre variables : un identifiant `id`, une estimation `load` de la charge processeur nécessaire pour terminer la tâche, une liste `t_c` de contraintes temporelles à respecter et `ca_oi`, l'instruction opératoire qui est l'objet de la requête et qui est déterminée par l'APRC quand aucune inconsistence n'a été trouvée dans la détermination de l'ordonnancement. L'agent attend une réponse de l'APRC pendant `r_timeout` tops d'horloge. La réponse peut être une acceptation ou un refus. Dans le premier cas, l'agent peut

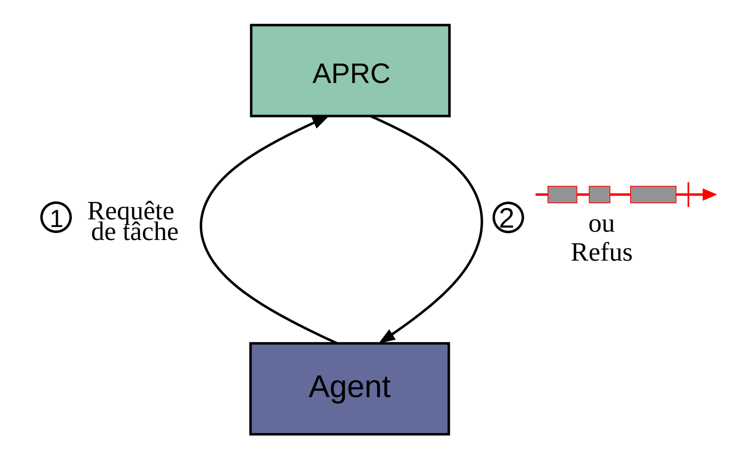


FIG. 4.5 – Protocole d'interaction sans conflit.

```
Def_CA_Management :=
  !request(r_content);T(r_timeout);
  ((?accepted(r_content);((r_content.ca_oi+0)||Def_CA_Management)+
   (?rejected(i_rejection));
     Def_CA_Management));
  Def_CA_Management
```

FIG. 4.6 – Instruction opératoire de base de l'APRC.

décider ou non de suivre l'instruction opératoire `r_content.ca_oi` créée par l'APRC. Dans tous les cas, un nouvel appel à `Def_CA_Management` permet à l'agent de faire une nouvelle requête auprès de l'APRC.

4.1.7 Gestion des conflits

Nous considérons dans cette partie qu'un agent a demandé à l'APRC d'allouer des tranches de temps pour une tâche, mais que celui-ci a refusé.

4.1.7.1 Propriétés de la solution proposée

L'agent demandeur peut, après le refus de l'APRC d'allouer une nouvelle tâche, décider de négocier avec les autres agents pour qu'ils sacrifient une part des tranches de temps qui leur ont été allouées pour leurs artifacts. L'APRC jouera le rôle d'intermédiaire dans ce protocole de négociation.

4.1.7.2 Extension des compétences de l'APRC

Nous étendons les compétences de l'APRC pour qu'il soit le support du protocole d'interaction entre les agents. Le protocole est décrit de manière informelle dans la figure 4.7.

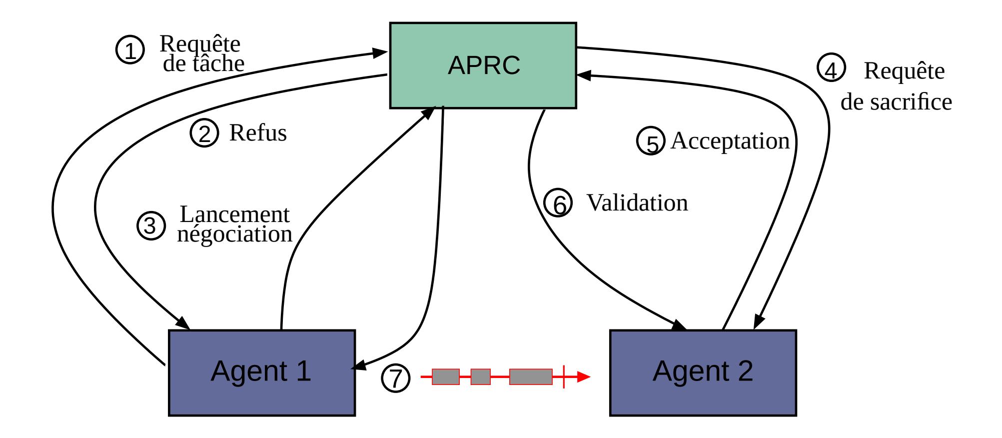


FIG. 4.7 – Protocole d'interaction avec conflit.

Le mode d'emploi de l'APRC est étendu pour refléter la nouvelle utilisation. La figure 4.8 donne la nouvelle version des instructions opératoires.

```
Def_CA_Management :=
  !request(r_content);T(r_timeout);
  ((?accepted(r_content);((r_content.ca_oi+0)||Def_CA_Management)+
   (?rejected(i_rejection);Def_Negotiation_Initiator(i_rejection));
     Def_CA_Management));
  Def_CA_Management

Def_Negotiation_Initiator(i_rejection) :=
  !start_negotiation(i_negotiation, i_rejection);  T(n_timeout);
    (?negotiation_failed+
     ?negotiation_succeeded(i_negotation))

Def_Negotiation_Participant :=
  !get_sacrifice_request;
  (?finished+(?new_sacrifice_request(s_content);
    (T(s_timeout)+
     !refuse(s_content)+
     (!propose_sacrifice(s_content);
       (?commit_sacrifice(s_content)+
        ?cancel_sacrifice(s_content))))));
  Def_Negotiation_Participant
```

FIG. 4.8 – Instructions opératoires de l'APRC.

`Def_CA_Management` est l'instruction opératoire utilisée par les agents pour interagir avec l'APRC lorsqu'ils ont un travail à donner à un artifact de calcul. `Def_Negotiation_Initiator` est l'instruction opératoire utilisée si une inconsistence est détectée par l'APRC lors de l'ajout des

Action	Précondition
request	B possible(r_content) & ¬B have_oi(r_content)
start_negotiation	B work_importance(r_content.id, s_content.imp) & B remain_work_load(r_content, s_content.load)
get_sacrifice_request	-
refuse	B ¬more_important(s_content, my_work)
propose_sacrifice	B more_important(s_content, my_work) & B can_sacrifice_work_load(s_content.load)

Perception	Effet
accepted	B have_oi(r_content.ca_oi)
rejected	-
negotiation_failed	B ¬possible(r_content)
negotiation_succeeded	B have_oi(i_negotiation.r_content)
new_sacrifice_request	-
commit_sacrifice	-
cancel_sacrifice	-

FIG. 4.9 – Lien sémantique entre les IO de l'APRC et l'état mental de l'agent.

contraintes pour la nouvelle tâche dans l'agenda. Dans ce cas, les autres agents s'engagent dans l'instruction opératoire `Def_Negotiation_Participant` pour éventuellement sacrifier une part des ressources processeur qu'ils utilisent.

Def_CA_Management. Nous avons apporté une légère modification à la version présentée dans la section 4.1.6. Quand un refus est reçu, l'agent essaie de négocier avec les autres agents en déclenchant l'instruction opératoire `Def_Negotiation_Initiator`.

Def_Negotiation_Initiator. Pour simplifier le processus de négociation, nous considérons que les agents ont un moyen commun pour exprimer l'importance d'une tâche de calcul. Cela permet à un agent qui reçoit une requête pour sacrifier une partie de ses ressources processeur de pouvoir comparer l'importance de ses tâches avec l'importance des tâches pour lesquelles on lui demande de se sacrifier. Le terme `i_negotiation` contient les informations de `r_content` et de `imp`, calculées par l'agent. Celles-ci représentent l'importance de la nouvelle tâche. L'agent demandeur exécute simplement l'action `start_negotiation` et attend une réponse pendant `n_timeout` tops d'horloge. Si la négociation a été fructueuse, l'APRC génère une instruction opératoire pour contrôler l'artifact de calcul et l'affecte à `ca_oi` dans `i_negotiation`. Notons que l'initiateur n'a aucune vue du processus de négociation : celui-ci est entièrement géré par le comportement interne de l'APRC.

Def_Negotiation_Participant. Un participant au protocole de négociation exécute l'action `get_sacrifice_request` de manière récursive. Les perceptions associées à cette action sont : `finished`, qui indique que la participation de l'agent au protocole est terminée ; et `new_sacrifice_request`, qui indique qu'une nouvelle requête de sacrifice `s_content` est

disponible. Le terme s_content contient les informations de i_negotiation et également l'information s_result, calculée par le participant, sur le sacrifice qu'il est capable de consentir. Quand une nouvelle requête de sacrifice est reçue, l'agent dispose de s_timeout tops d'horloge pour répondre. Il peut refuser de se sacrifier ou il peut accepter, et dans ce cas, il doit fournir une proposition qui sera utilisée par l'APRC pour vérifier si, avec le sacrifice, la nouvelle tâche peut s'exécuter. Selon le cas, l'APRC demande à l'agent de valider ou d'annuler le sacrifice proposé.

Les instructions opératoires, données seules, ne sont pas suffisantes pour les agents. Il faut en effet donner aux agents suffisamment d'informations pour comprendre les instructions opératoires qu'ils exécutent. Nous donnons donc un lien entre les instructions opératoires et l'état mental de l'agent. La figure 4.9 décrit ce lien. Nous utilisons la sémantique mentale liée aux actions et perceptions introduite dans [Viroli et Ricci, 2004] et rappelée dans la section 3.4.3.4.1. Dans ce tableau, les croyances de l'agent permettent d'exprimer des préconditions aux actions des instructions opératoires et les perceptions ont des effets sur les croyances de l'agent.

Préconditions aux actions. Un agent envoie une requête à l'APRC s'il croit qu'il est possible de réaliser la tâche (Bpossible(r_content)) et s'il ne dispose pas encore d'une instruction opératoire pour exécuter cette tâche (¬Bhave_oi(r_content)). Avant de démarrer une négociation, il remplit des champs dans le terme s_content (l'importance et la charge de calcul de la tâche). Un agent refuse de se sacrifier s'il croit que ses travaux sont plus importants que celui associé à la requête (B¬more_important(s_content, my_work)). Au contraire, il propose de se sacrifier s'il croit que ses tâches sont moins importantes. La proposition qu'il formule dans ce cas dépend de ses croyances sur la quantité de ressources de calcul qu'il peut sacrifier (Bcan_sacrifice_work_load(s_content.load)).

Effets des perceptions. Si un agent reçoit la perception que sa requête a été acceptée ou que la négociation a été fructueuse, il met à jour sa base de croyances pour croire que r_content.ca_oi est une instruction opératoire valide qu'il va pouvoir utiliser pour réaliser sa tâche de calcul (Bhave_oi(r_content)). Dans le cas où la négociation échoue, il met à jour son état mental pour ne plus croire qu'il peut réaliser la tâche demandée (B¬possible(r_content)).

4.1.8 Positionnement par rapport aux systèmes temps réel

Nous proposons un ordonnanceur qui permet de respecter des délais pour les différentes tâches exécutées sur le système. Or, c'est justement un des rôles du système d'exploitation que de gérer l'ordonnancement des tâches sur le ou les processeurs disponibles. Cela est fait différemment selon les propriétés recherchées. Ainsi, des systèmes comme Windows ou Unix, dits à temps partagé, garantissent que tous les processus pourront disposer régulièrement d'un quantum de temps processeur. Ce qui est recherché ici est l'illusion, au niveau d'abstraction de l'utilisateur, que tous les processus s'exécutent en même temps. Malheureusement, ces systèmes ne garantissent rien sur les dates de fin des tâches.

Cette garantie est en revanche fournie par les systèmes temps réel, mais cela est souvent obtenu au prix de la perte de l'illusion que les programmes sont toujours actifs. En effet, les algorithmes classiquement utilisés comme Earliest Deadline First proposé dans [Liu et Layland, 1973] préfèrent lancer une tâche jusqu'à ce qu'elle soit terminée et passer ensuite à la suivante plutôt que d'assigner de petits quanta de temps à chaque tâche.

Nous détaillons ici des raisons supplémentaires qui nous poussent à nous détourner des systèmes d'exploitation temps réel pour rester sur des systèmes d'exploitation plus classiques et ajouter les fonctionnalités manquantes sous forme de services fournis par l'APRC.

4.1.8.1 Hypothèses de départ

Les hypothèses de départ sont très différentes dans notre problème et dans ceux résolus par les systèmes temps réel. Ces derniers se basent la plupart du temps sur le temps d'exécution au pire des cas des algorithmes (WCET : Worst Case Execution Time) et sur des tâches périodiques. Cela permet de calculer un ordonnancement une fois pour toutes et d'être certain que toutes les tâches pourront respecter leurs délais.

Dans notre cas, où nous utilisons des algorithmes issus de l'Intelligence Artificielle, le temps d'exécution au pire des cas est généralement énorme et très éloigné du temps d'exécution moyen. De plus, il est généralement irréaliste de considérer que les tâches exécutées dans un système multi-agent sont périodiques. Cependant, comme nous n'utilisons pas les hypothèses de base des systèmes temps réel, nous ne pouvons garantir que toutes les tâches pourront s'exécuter. Nous sommes donc obligés de concevoir nos agents de telle manière qu'ils pourront toujours continuer leur travail même si les tâches qu'ils voulaient déléguer à des artifacts de calcul ne peuvent s'exécuter.

4.1.8.2 Contraintes au niveau des systèmes

Les contraintes sur la programmation des processus sur lesquels on pose des contraintes temporelles sont également très fortes sur les systèmes temps réel. On ne peut par exemple réaliser aucune opération pour laquelle on ne peut connaître très précisément le nombre de cycles processeur nécessaires. C'est le cas de toutes les opérations d'entrée/sortie. Ainsi, il est impossible pour un processus temps réel d'effectuer directement un affichage sur l'écran ! Pour cela, il faut créer un second processus sans contraintes temps réel qui va partager la mémoire du processus temps réel et va être capable d'interagir avec les périphériques d'entrée/sortie.

Ces contraintes sont très dures et irréalistes dans le cadre de programmes issus du domaine de l'Intelligence Artificielle. Nous voulons pouvoir utiliser dans nos SMA toutes les facilités proposées par les bibliothèques de programmation proposées sur les systèmes non temps réel classiques.

4.1.9 Critique

4.1.9.1 Implications sur la modélisation de l'application

La gestion des ressources processeur que nous proposons implique une structuration très stricte des applications. Nous obligeons par exemple que les tâches ayant des délais soient déléguées à des artifacts de calcul, même si ce sont des tâches courtes que l'agent aurait pu réaliser lui-même sans perdre sa propriété d'extraversion.

Nous avons également considéré jusqu'à maintenant que toutes les tâches déléguées aux artifacts disposent d'un délai. Il se peut que dans certaines applications, ceci ne soit pas réaliste et oblige à définir des délais de manière fictive. La relaxation de cette contrainte est discutée dans la section 4.2.

Comme nous voulons préserver l'autonomie de décision des agents, nous ne pouvons déléguer entièrement toutes les questions d'ordonnancement à l'APRC. Cela implique que quand on conçoit un SMA, on doit prévoir pour chaque agent :
- soit qu'il sera capable de continuer à travailler sans avoir pu réaliser une de ses tâches de calcul,
- soit qu'au moins un autre agent du système va s'être sacrifié pour qu'il puisse réaliser sa tâche.

4.1.9.2 Efficacité pratique de l'algorithme d'ordonnancement

Nous avons étudié les limites de l'ordonnanceur. La complexité provient principalement du découpage en sous-intervalles, qui est en $O(2^n)$. Le temps de calcul devient très vite prohibitif. Il est cependant possible en pratique de faire baisser cette complexité en étudiant les contraintes posées. En effet, il est souvent possible de prédéterminer, après le calcul de ESR_i et des $C_{i,k}$ qu'un certain nombre de tâches ne pourront pas du tout s'exécuter dans l'intervalle i considéré. Si la somme $ESR_i+ER_{i,i}$ est égale à la durée de l'intervalle i, il n'est pas nécessaire de considérer d'autres tâches que la tâche i et les tâches k pour lesquelles $C_{i,k} = 1$.

Par exemple, dans le cas dégénéré où la tâche i ne peut tenir sa date limite qu'en prenant toutes les ressources processeur du début à la fin de l'intervalle, il ne sert à rien de faire le calcul de l'ordonnancement sur cet intervalle en considérant d'autres tâches.

De plus, il est nécessaire de limiter le nombre de tâches dans un intervalle pour que la solution trouvée dans celui-ci n'apporte pas un trop grand fractionnement. Il doit subsister une différence entre les ordres de grandeur des périodes de temps considérées par l'ordonnanceur système et celui de l'APRC. Ce dernier ne doit pas empiéter sur le travail du premier en mettant en pause et en redémarrant trop souvent les agents. Nous limitons donc actuellement le nombre de tâches dans un intervalle à 10 au maximum. Ainsi, il se peut que nous ne trouvions pas d'ordonnancement alors qu'il en existe réellement un. Ceci n'est pas véritablement un problème car de toutes façon, si nous déterminions l'ordonnancement possible, il ne serait pas tenu en pratique à cause de l'empiètement sur le travail de l'ordonnanceur du système d'exploitation.

4.1.9.3 De la centralisation

L'APRC apporte un point de centralisation. Nous avons déjà discuté du bien fondé de son introduction dans la section 4.1.1. Nous avions remarqué qu'une solution totalement décentralisée serait très difficile à mettre en place et serait certainement inefficace.

Il est important de vérifier que la centralisation que nous avons faite ne nuit pas à l'aspect agent de nos applications. Nous voulons donc insister sur le fait que nous n'avons centralisé que les informations nécessaires au calcul de l'ordonnancement, mais que nous n'avons pas centralisé les décisions liées à l'ordonnancement. Les agents gardent l'entier contrôle sur le sacrifice éventuel de leurs ressources de calcul qu'il peuvent avoir à faire.

4.2 Gestion des ressources processeur sans délais à respecter

Notre cadre est celui où l'on donne une date limite à chaque tâche de calcul déléguée à un artifact. Cependant, lorsque les tâches que l'on doit exécuter dans un SMA ne sont pas soumises à des délais stricts, on peut quand même continuer à bénéficier des services de l'APRC si ponctuellement un agent a un besoin un peu plus important en ressources processeur. Notre APRC est prévu pour fonctionner en mode simplifié dans ce type d'applications. On ne passe plus par la première partie du protocole qui permet de déterminer un ordonnancement pour une nouvelle tâche. On utilise uniquement les facilités de négociation proposées par l'APRC. Ainsi, un agent pourra décider de stopper ses calculs pendant quelques temps pour laisser l'autre finir plus rapidement ses calculs.

Ce mode de fonctionnement peut permettre de gérer très facilement des priorités d'accès aux ressources processeur entre les différents agents, et ceci de manière décentralisée. Les différents niveaux de priorité entre les agents sont définis au niveau de chaque agent. Un agent qui reçoit une demande de sacrifice pour un autre agent plus prioritaire que lui accepte toujours et il refuse toujours si la demande provient d'un agent moins prioritaire.

4.3 Passage au multi-processeur

Notre modèle peut être étendu à plusieurs processeurs. Nous plaçons pour cela un APRC sur chacun des processeurs du système. Chaque APRC s'occupe de l'allocation des ressources du processeur sur lequel il s'exécute. Ils sont également chargés de mesurer la charge de leur processeur de différentes manières :

1. Pour les tâches planifiées, le rapport $\frac{E*P}{(DL-T_0)}$ où E est le nombre d'opérations à affecter à la tâche, P la puissance du processeur en nombre d'opérations par secondes et $(DL-T_0)$ l'intervalle de temps autorisé pour l'exécution de la tâche. Ce rapport permet de mesurer la marge de manœuvre dont dispose l'APRC pour planifier d'autres tâches.

2. Pour les agents, le taux de requêtes de tâches qui aboutissent à un refus sur un intervalle de temps donné.
3. Le nombre d'agents et le nombre de tâches planifiées à partir de la date courante.

Ces informations sont partagées entre tous les APRC à intervalles réguliers ou sur demande de l'un d'eux.

Un agent peut demander à tout moment à son APRC s'il lui serait favorable de migrer sur un autre processeur. L'APRC remet à jour ses informations sur les autres processeurs et propose à l'agent de migrer sur le processeur qui dispose des meilleures mesures de charge, l'ordre de préférence étant l'ordre dans lequel nous avons défini les mesures.

L'APRC peut également prendre l'initiative d'indiquer à un agent auquel il vient de refuser l'exécution d'une tâche qu'un autre processeur moins chargé pourrait l'accueillir. Cela sera fait uniquement si le taux de requêtes refusées pour cet agent dépasse un seuil fixé à l'avance.

Lorsqu'un agent a migré, il attend la confirmation de l'APRC avant de poursuivre son exécution, tout comme nous l'avons vu dans la section 4.1.4.1 lorsque nous parlions de la création de nouveaux agents et de leur prise en compte dans l'ordonnancement. Un nouvel arrivant sur un processeur doit également se synchroniser avec l'horloge « temps APRC » de celui-ci pour que les requêtes de tâche qu'il va effectuer à l'avenir soient correctement datées. Avec ce mécanisme, nous n'avons donc pas besoin d'horloge globale pour l'ensemble des processeurs.

Nous verrons dans la section 5.4.2 un exemple d'utilisation de la migration dans lequel tous les agents débutent leur exécution sur un seul processeur puis quelques-uns d'entre eux migrent vers un second processeur jusqu'à ce que la charge des deux soit comparable.

4.4 Synthèse

Nous avons posé dans ce chapitre le problème du respect de contraintes temporelles sur les calculs longs que les agents délèguent aux artifacts.

Nous avons proposé un algorithme d'ordonnancement basé sur l'algorithme du simplexe. Il est utilisé par un artifact de coordination de l'accès aux ressources processeur nommé APRC pour répartir les ressources entre les agents et les artifacts. Les agents font obligatoirement appel à l'APRC pour déléguer une partie de leurs traitements longs sous forme de tâches de calcul dont les durées sont déterminées par l'agent. L'APRC permet de garantir que les délais sur les tâches de calcul des artifacts seront respectées, tout en préservant l'accès permanent des agents au processeur.

Nous avons détaillé le mode d'emploi de l'APRC. Celui-ci permet aux agents de réaliser des requêtes pour de nouvelles tâches. Si celles-ci ne peuvent être satisfaites, l'APRC préserve l'autonomie des agents en refusant la nouvelle tâche. L'agent demandeur peut dans ce cas utiliser l'APRC comme intermédiaire dans la négociation avec les autres agents pour qu'ils sacrifient un part des ressources

processeur qui leur ont été allouées. Nous avons également vu comment il était possible de bénéficier des services de l'APRC dans des SMA où les agents n'ont pas de délais à respecter mais ont par moments des besoins plus importants en ressources processeur.

Le modèle de base fonctionne avec un APRC pour gérer un unique processeur. Nous pouvons très facilement étendre ce modèle à plusieurs processeurs en utilisant un APRC par processeur. Dans ce nouveau cadre, il paraît intéressant de donner la possibilité de lancer un nouvel agent ou un nouvel artifact sur un autre processeur si celui sur lequel on travaille devient surchargé. La migration d'agents et d'artifacts est également une possibilité de la plate-forme multi-agent Alba que nous utilisons (et qui est décrite dans le chapitre suivant). Un agent devrait donc disposer des informations nécessaires à la décision de migrer sur un autre processeur. Nous étendons pour cela les compétences de l'APRC pour qu'il soit capable de mesurer de différentes manières la charge du processeur qu'il gère, qu'il échange ces informations avec les autres APRC du système et qu'il puisse donner des indications aux agents sur les processeurs vers lesquels ils peuvent migrer.

Chapitre 5

Mise en œuvre et évaluation

Évaluer, c'est créer : écoutez donc, vous qui êtes créateurs ! C'est l'évaluation qui fait des trésors et des joyaux de toutes choses évaluées.
Friedrich Nietzsche.

Ce chapitre est consacré à la description de l'implémentation des propositions des chapitres précédents sur notre plate-forme multi-agents ; Cela nous permettra de décrire comment nous avons évaluée nos propositions et quels sont les résultats de cette évaluation.

5.1 Mise en œuvre

Nous avons mis en place nos artifacts de calcul et l'APRC décrits précédemment dans la plateforme multi-agents développée au sein de Thales Division Aéronautique. Celle-ci est constituée d'ALBA, une librairie générique pour la création d'agents mobiles en Prolog et de différentes autres bibliothèques pour mettre en œuvre le modèle d'agent choisi ainsi que les protocoles de communication entre les agents. Nous décrivons dans les paragraphes suivants les principales caractéristiques d'Alba, le modèle d'agent que nous utilisons actuellement et la manière dont nous y avons intégré les artifacts.

5.1.1 ALBA

5.1.1.1 Une plateforme pour Prolog

La première exigence quand il a été question de déployer une plateforme multi-agents a été de pouvoir implémenter des agents en Prolog. En effet, il était très important de pouvoir réutiliser l'importante base de code Prolog qui a été constituée au fil des années de recherche dans le domaine de l'Intelligence Artificielle au sein de Thales.

Nous pensons également que Prolog possède un grand nombre de qualités nécessaires à un langage de programmation de SMA :

- l'accès aux mécanismes système comme les sockets, la gestion des processus, les entrées/sorties sont maintenant correctement gérées par la plupart des Prolog du marché,
- l'introspection peut être réalisée grâce aux facilités de modifications dynamique du code,
- Prolog permet aussi bien la programmation déclarative que procédurale,
- son efficacité naturelle permet de développer et de tester très rapidement de nouveaux proptotypes et idées,
- sa nature interprétée lui permet d'être exécuté sans modification sur une large variété de plateformes.

5.1.1.2 Caractéristiques principales

ALBA est une bibliothèque Prolog dédiée à la programmation d'agents écrits en Prolog. Les fonctionnalités d'ALBA sont décrites en détail dans [Devèze *et al.*, 2006]. Nous présentons ici celles qui nous paraissent essentielles.

5.1.1.2.1 Décentralisation

Nous avons remarqué que la plupart, sinon la totalité, des plateformes agents qui implémentent des agents mobiles utilisent une forme de centralisation (que ce soit par un serveur fournissant la gestion des agents pour chaque machine/contexte ou par un serveur central gérant l'ensemble des agents distribuées sur différents ordinateurs). Dans tous les cas, la migration, la création des agents et les communications sont réalisées au travers d'une entité dédiée qui connaît les agents locaux et leurs équivalents distants.

Notre plateforme est entièrement décentralisée. Nous n'utilisons pas d'intergiciel auquel tous les agents devraient se connecter. Au contraire, le code implémentant les fonctionnalités de la plateforme est embarqué dans chacun des agents. La figure 5.1 illustre l'architecture très simple utilisée pour nos agents ALBA.

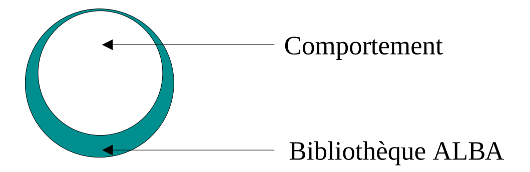


FIG. 5.1 – Un agent ALBA.

5.1.1.2.2 Généricité

ALBA se veut générique : il n'est fait aucune hypothèse concernant le modèle d'agent utilisé par les applications développées avec ALBA. Ainsi, ALBA peut être utilisée avec n'importe quel type de modèle d'agent (Agent-0, AgentSpeak, BDI, 3APL, ...). De plus, la communauté agent n'a pas encore proposé de modèle universel qui pourrait être utilisé pour tous les types d'applications possible. Cette approche nous permet de pouvoir tester différents modèles et de les combiner pour atteindre nos buts. La même idée est utilisée pour ne pas être dépendant du langage de communication utilisé par les agents sur les canaux de communication ouverts par la bibliothèque ALBA.

5.1.1.2.3 La migration des agents

Pour autoriser la migration, nous plaçons sur chaque machine du SMA un démon ALBA. Les démons sont chargés d'exécuter sur les machines distantes les primitives de création ou de migration appelées par un agent et de gérer les transferts de données éventuellement nécessaires. La continuité du service de messagerie est également assurée pour que les agents ne perdent pas de messages pendant le processus de migration.

La figure 5.2 illustre le fonctionnement de la migration d'agents ALBA d'une machine à une autre.

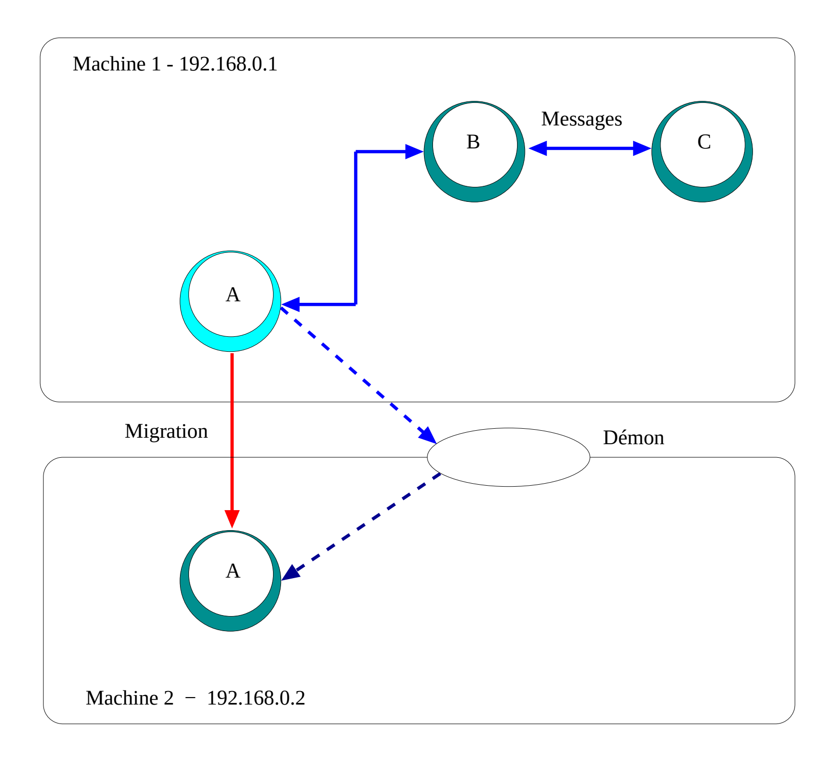


FIG. 5.2 – La migration des agents dans ALBA.

5.1.2 Modèle d'agent utilisé

Les différentes version de la plateforme de développement d'agent de Thales ont essayé de décoreller la partie architecturale du sous-système d'exécution et le modèle d'agent réellement implanté dans les applications. Le but est de pouvoir tester plusieurs modèles et construire de manière incrémentale celui qui convient aux applications de Thales.

Les premières applications ont donc été codées en Prolog pur, sans aucune bibliothèque spécifique au développement agent. Un certain nombre de limitations sont rapidement apparues, comme la difficulté à maintenir différents contextes de conversation et de raisonnement de manière simultanée et ceci sans utiliser de processus légers car l'implémentation de Prolog que nous utilisons ne les fournit pas.

Mon stage de DEA ([Dinont, 2003]) a consisté à proposer un modèle d'agent et un langage sous forme de bibliothèque Prolog pour gérer ces aspects. Le modèle proposé est à base d'automates temporisés. Les agents peuvent instancier différents automates pour leurs différentes conversations. Les messages sont aiguillés vers le bon automate en fonction d'un identifiant de conversation défini par l'initiateur d'une conversation. Une des spécificités de ce modèle consiste à être obligé de spécifier un timeout et l'action correspondante pour chaque transition de l'automate correspondant à une attente de message. J'ai également proposé une conversion automatique du fonctionnement global des agents utilisant ce modèle en réseaux de Pétri colorés.

Après mon DEA, le modèle a continué à évoluer. Il est actuellement basé sur la notion de Fil De Raisonnement (FDR). La figure 5.3 illustre l'architecture d'un agent utilisant les FDR.

Chaque FDR peut être vu comme un contexte. Un modèle de FDR est décrit comme une machine à états finis étendue représentant des connaissances procédurales associées à un contexte. Quand un FDR est instancié, une mémoire locale est également créée. Elle permet de stocker les données relatives à ce contexte. De plus, une mémoire globale est partagée par les différents FDR de l'agent.

Tous les messages arrivants de l'extérieur sont traités par un aiguilleur qui est chargé de dispatcher les messages aux bons FDR en fonction d'un ensemble de règles de grammaire qui peuvent prendre en compte l'expéditeur, la syntaxe et le contenu des messages entrants. Si un message ne peut être filtré par l'aiguilleur, il est automatiquement envoyé au FDR par défaut. Celui-ci est principalement chargé d'identifier de nouveaux contextes et de gérer les messages inconnus.

Les FDR sont instanciés et détruits dynamiquement selon l'évolution du système. Les règles de grammaires de l'aiguilleur et les modèles de FDR peuvent également être ajoutés, supprimés ou modifiés pendant que l'agent s'exécute, permettant ainsi d'adapter dynamiquement le comportement de l'agent.

Ce modèle d'agent propose une vision concurrente des différents contextes d'exécution, ceci sans utiliser le mécanisme des processus légers. Cela permet d'éviter l'apparition d'indéterminisme comme nous l'avions évoqué dans la section 2.2.2.1.1.

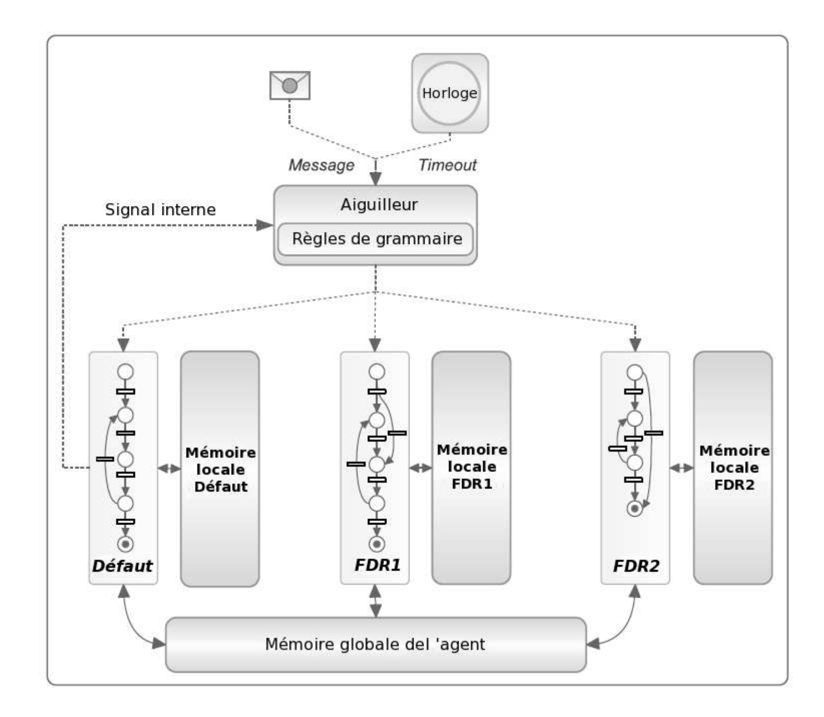


FIG. 5.3 – L'aiguilleur dispatche les messages vers les fils de raisonnement.

Lorsque l'agent démarre, l'aiguilleur, le FDR par défaut et la mémoire globale sont initialisés. Un interpréteur est chargé de l'exécution du programme de l'agent. Il conserve les informations relatives aux FDR comme leur nom, leur état courant, les valeurs des timeouts posés et leur mémoire locale. Il conserve également une file des événements non encore traités. Un cycle d'exécution est décrit ainsi :

```
TANT QUE il reste au moins une instance de FDR FAIRE
  calculer le timeout à appliquer
         (minimum des timeouts restants des FDR instanciés)
  lire les messages entrants pendant au maximum timeout secondes
  SI il y a des messages à traiter ALORS
    appliquer la procédure stratégie_de_filtration
          pour sélectionner le message M à traiter
    appliquer les règles de grammaire sur M pour déterminer le FDR à réveiller
  SINON
    réveiller les FDR pour lesquels le timeout a expiré
  FIN SI
  exécuter séquentiellement les actions des transitions qui ont été activées
  supprimer les instances de FDR qui sont arrivées dans l'état FIN
FIN TANT QUE
```

5.1.3 Intégration des artifacts à la plateforme

Nous avons étudié et implémenté l'intégration des artifacts de calcul présentés dans cette thèse au sein de notre plateforme, qui était initialement conçue pour n'accueillir que des agents.

Paul-Édouard Marson a proposé durant son master recherche [Marson, 2006] une bibliothèque qui complète celle d'ALBA et des fils de raisonnement pour la création d'artifact. Ceux-ci utilisent la bibliothèque ALBA pour communiquer avec les agents. Les instructions opératoires sont décrites sous la forme de prédicats Prolog. Chaque instruction opératoire dispose d'un identifiant unique. Le comportement interne des artifacts est implémenté sous forme de FDR. La grammaire lie l'identifiant d'une instruction opératoire avec le FDR correspondant. Ainsi, lorsqu'une action est demandée par un agent pour une instruction opératoire donnée, le message est automatiquement envoyé au bon FDR.

La nouvelle bibliothèque intègre également les outils nécessaires pour que les agents puissent utiliser les artifacts. Paul-Édouard Marson a mis au point un mécanisme de conversion automatique d'instructions opératoires vers des fils de raisonnements. Ainsi, le pilotage d'un artifact se fait de manière transparente au travers d'un contexte quelconque de l'agent.

5.2 Présentation de l'application Interloc

Les artifacts ont été utilisés dans diverses applications développées chez Thales Division Aéronautique. Cela concerne par exemple la planification de mission ou encore la coordination d'un ensemble de drones. Il apparaît même que tous les SMA que nous avons développés à ce jour ont mis en évidence la nécessité de faire cohabiter des agents cognitifs autonomes et des agents qui, bien que s'exécutant en parallèle et ne dépendant des autres que par des échanges de messages, ne pouvaient en aucun cas être considérés comme autonomes, la spécification de leur comportement étant connue lors de la programmation des autres agents. A la différence des exemples traités par l'équipe de Bologne, nos artifacts concernaient moins la coordination que la mise en œuvre d'algorithmes complexes tels que ceux issus des recherches en Intelligence Artificielle (propagation de contraintes, recherche heuristique, ...) que nous ne voulions pas voir traités directement par les agents afin de ne pas les bloquer si la durée des traitements devenait prohibitive, ce qui se produit assez souvent avec de tels algorithmes.

Parmi ces applications, l'une d'entre elles mérite une mention particulière. : Interloc (figure 5.4). Elle concerne la localisation de bateaux à partir d'avions d'observation par la seule utilisation de la direction relative du radar de veille du bateau observé. Cette méthode permet à l'avion de rester discret et donc d'éviter que le bateau prenne conscience du repérage auquel il est soumis. Elle peut être utilisée pour le repérage de pétroliers opérant un dégazage illicite et bien entendu également dans de nombreuses applications de nature militaire. Le principe est simple : grâce à un capteur spécifique (analogue à un goniomètre) l'avion mesure, à intervalles réguliers correspondant à la période du radar du bateau, la direction de ce dernier par rapport à l'avion. Les angles mesurés sont imprécis et peuvent même manquer en raison des conditions ambiantes difficiles prévalant dans ce type d'application. Si le bateau était immobile, une simple triangulation permettrait très rapidement de le localiser. Mais ce n'est bien sûr pas le cas. Néanmoins le calcul n'est possible que si la vitesse du bateau est notablement inférieure à celle de l'avion (un facteur 10 par exemple) et si la trajectoire de l'avion est correctement choisie, conditions en général faciles à satisfaire. Le calcul est effectué par un algorithme

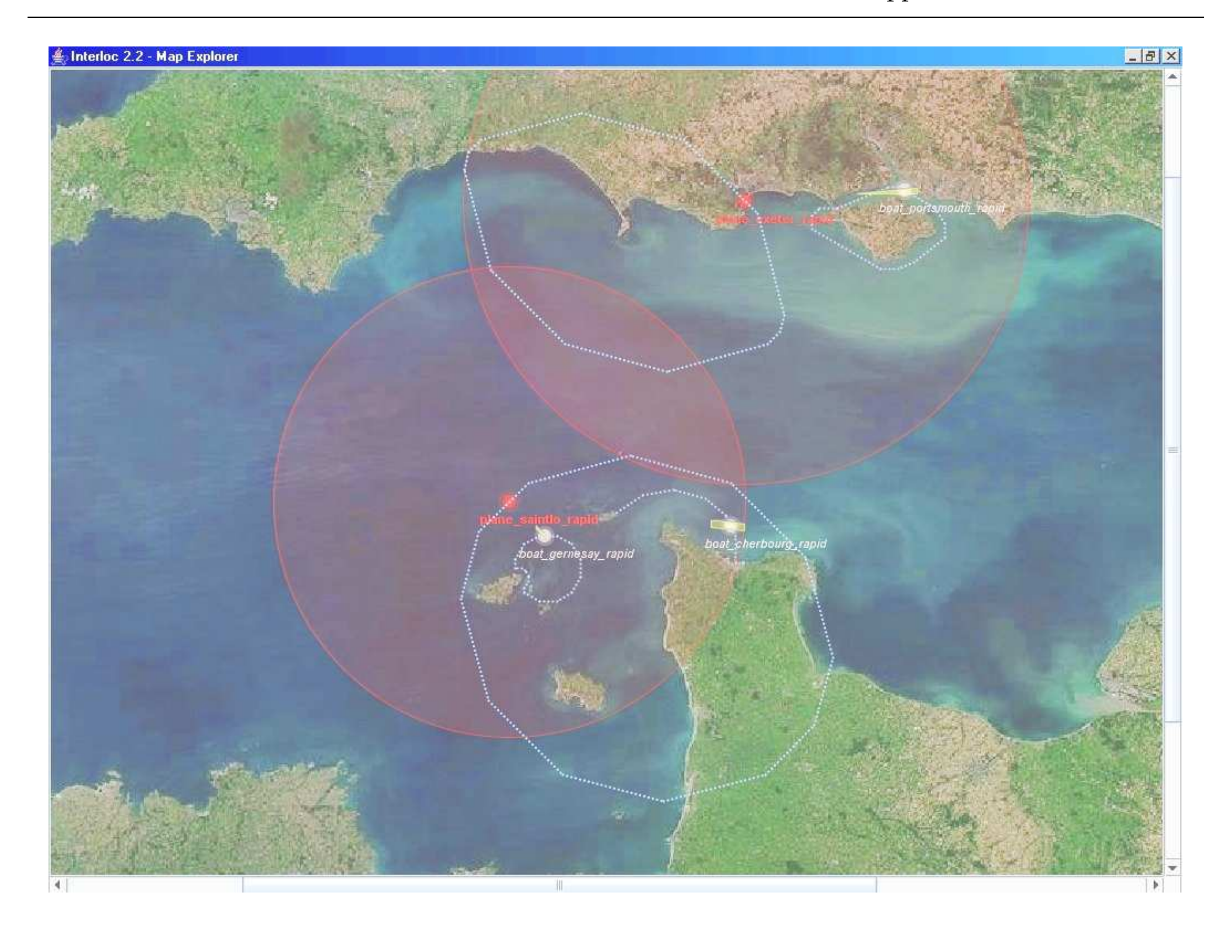


FIG. 5.4 – Copie d'écran d'Interloc.

de propagation d'intervalles sur les domaines continus [Lhomme, 1993] qui exploite la différence de vitesse entre bateaux et avion pour fournir une localisation qui est progressivement réduite au fur et à mesure que les données arrivent. La propagation d'intervalle est particulièrement bien adaptée à ce genre de calcul car d'une part elle permet d'injecter de nouvelles contraintes de manière incrémentale (contraintes établies à partir des mesures reçues), et d'autre part de prendre en compte des contraintes non linéaires (en l'occurrence trigonométriques).

Dans Interloc, nous utilisons un propagateur de contraintes sur intervalles qui a été développé et mis en œuvre depuis de nombreuses années (Interlog [Botella et Taillibert, 1993]). Il est possible de lui spécifier un contrat de temps à chaque fois qu'une nouvelle contrainte est introduite. Si le point fixe n'a pas été atteint lorsque le contrat de temps expire, Interlog s'arrête et retourne un ensemble d'intervalles spécifiant la localisation obtenue à ce stade du traitement. L'agent peut alors décider de relancer la propagation pour un nouveau contrat de temps afin d'améliorer la précision de la localisation ou, au contraire, d'introduire les contraintes correspondant à une mesure nouvellement arrivée. Pour prendre une telle décision, il estime l'amélioration apportée à la localisation par la réduction de sa surface obtenue au cours d'un contrat de temps.

De manière plus formelle, le problème est le suivant : Soient B_i les bateaux, dont le nombre peut varier en fonction du temps. Soit p_i la période du radar du bateau B_i. Quand une nouvelle mesure j arrive pour le bateau B_i, une nouvelle tâche $T_{i,j}$ est créée. Son délai est : $DL_{i,j} = T + \tau_{i,j}$ avec T la date d'arrivée de la mesure et $\tau_{i,j}$ la durée du calcul à réaliser.

Comme peu de choses sont connues sur la durée du calcul pour converger vers la valeur finale, la même valeur PD est utilisée comme contrat de temps pour le propagateur de contraintes pour chaque nouvelle tâche. Donc : $\forall_i \forall_j : \tau_{i,j} = PD$.

A la fin du contrat, un résultat est toujours disponible car, grâce à la monotonie de la procédure de rétrécissement de la zone de localisation utilisé par le propagateur sur intervalles, il y a à tout moment un intervalle valide contenant la position actuelle du bateau qui peut être retourné.

5.3 Nouvelle modélisation d'Interloc avec les artifacts

La gestion des temps de traitement est primordiale pour un agent utilisant le type d'algorithme décrit dans le paragraphe précédent. En effet, la durée des calculs peut changer de manière considérable car des phénomènes de convergence lente peuvent apparaître lors du processus de propagation [Lhomme *et al.*, 1994]. Pour conserver son autonomie, ne serait-ce que pour pouvoir décider d'abandonner la poursuite d'un bateau donné si celle-ci devient inutile, l'agent doit donc se protéger de blocages éventuels. La création d'un artifact de calcul est donc apparue comme une bonne solution car elle permettait de découpler la partie cognitive de l'agent de celle chargée d'effectuer les calculs, laissant à l'agent la possibilité d'une délibération pour décider d'arrêter un calcul devenu trop long ou de prendre en compte ou non une mesure nouvellement arrivée. L'artifact peut bien sûr être implémenté comme un agent ne possédant pas de capacité de décision propres ; c'est d'ailleurs ainsi que nous avions procédé initialement. Cependant, le type d'interaction régissant les rapports entre l'artifact et les autres agents est différent des rapports entre agents. Ainsi, comme nous l'avons montré dans [Dinont *et al.*, 2006a], il doit être possible de suspendre puis relancer l'exécution de l'artifact afin de pouvoir exploiter au mieux les ressources offertes par la machine hôte, ce qui n'est pas souhaitable pour un agent réellement autonome souhaitant pouvoir réagir aux changements de son environnement. Par ailleurs, le comportement des artifacts est connu à l'avance et la programmation de la délibération de l'agent peut donc en tenir compte, ce qui la rend notablement plus aisée.

Dans l'article [Dinont *et al.*, 2006a], nous avons montré comment le contrôle d'un tel artifact et l'utilisation d'un artifact de coordination permettait à la fois de garantir que l'agent était en mesure de décider de son comportement lors de l'arrivée d'une nouvelle mesure (contrat de temps terminé) et une meilleure utilisation de la puissance de calcul. Nous nous plaçons ici dans le cas où il n'y a pas d'artifact de coordination et donc où l'agent gère lui même ses relations avec l'artifact de calcul. Pour décrire ce dernier, nous utilisons le formalisme décrit dans la section 3.4.

Il faut tout d'abord décrire l'interface d'usage qui spécifie les actions et perceptions en direction et provenant de l'artifact. Dans notre application de localisation de bateaux, les actions possibles sur l'artifact de calcul sont :

– `add_constraint(C)` : introduction d'une contrainte C,
– `propagate`, `propagate(T)` : lancement de la propagation sans ou avec un contrat de temps T,
– `pause` : suspension du traitement,
– `restart` : redémarrage du traitement,
– `stop` : arrêt (normal) de l'artifact.

Par ailleurs, tout artifact peut être interrompu définitivement (destruction du processus ou du processus léger) mais cette action, au même titre que la création initiale, ne fait pas à proprement parler de l'interface d'usage.

Les perceptions possibles sont :
– `syntax_error` : perception d'une erreur de syntaxe si la contrainte introduite par l'action `add_constraint` est incorrecte,
– `success` : perception du succès de l'introduction d'une nouvelle contrainte,
– `result(E, I)` : fournie à la fin du contrat de temps et représentant l'état du calcul (E) et l'ensemble des intervalles de localisation (I) comprenant les coordonnées cartésiennes de la zone et ses coordonnées polaires relatives à la dernière position de l'avion.

L'élément essentiel nécessaire à la programmation de l'agent utilisant un artifact est le mode d'emploi de l'artifact (les instructions opératoires). Deux situations ont été considérées. La première correspond à une utilisation de l'artifact de propagation de contraintes sans contrat de temps. En cas de convergence lente, l'agent n'est pas bloqué mais le calcul continue jusqu'à son terme, qui peut être très lointain. L'agent a néanmoins la possibilité de « tuer » définitivement l'artifact. Le mode d'emploi correspondant est le suivant :

```
OI1 := !stop +
       (!add_constraint(C) ;
        ((?syntax_error ; OI1) +
         (?success ; (OI1 + (!propagate ; ?result(E,I); OI1)))))
```

Lorsque l'artifact est au repos, l'agent peut le stopper ou ajouter une nouvelle contrainte. Après l'ajout d'une contrainte, il peut percevoir qu'il y avait une erreur de syntaxe dans sa contrainte et dans ce cas, il reboucle sur la même instruction opératoire pour arrêter l'artifact ou lui reproposer une nouvelle contrainte. Dans le cas où la nouvelle contrainte est correcte, l'agent peut ajouter de nouvelles contraintes ou lancer le calcul. Il est important de remarquer qu'en disposant de cette instruction opératoire, l'agent peut déterminer qu'une fois le calcul lancé par l'action `propagate`, il recevra obligatoirement un résultat, mais dans un temps indéterminé.

La seconde situation correspond au cas où la résolution s'accompagne d'un contrat de temps au bout duquel l'algorithme renvoie une réponse même si le point fixe n'a pas été atteint. A cette réponse est associée une information sur l'état du calcul afin que l'agent puisse décider de la suite à donner. Cet état peut prendre trois valeurs :
– `disponible` : le dernier calcul demandé est terminé,
– `en cours` : le calcul n'est pas terminé, mais un minimum de temps processeur a été utilisé pour garantir une qualité minimum de la solution,

– `saturé` : dans le cas contraire.

Le mode d'emploi devient alors le suivant :

```
OI2 := !stop +
       (WORK_OI +
        (!add_constraint(C) ;
         (?syntax_error + ? success)) ; OI2)

WORK_OI := !propagate(T) ; (τ(T) + ?result(E,I))
```

Contrairement au premier cas, l'agent peut relancer le calcul, même sans avoir ajouté de contrainte. Précédemment, cette possibilité n'était pas nécessaire car le résultat obtenu était toujours le point fixe et donc relancer le calcul ne servirait à rien. Notons qu'en analysant ce nouveau mode d'emploi, l'agent peut déterminer que s'il lance la résolution avec un contrat de T secondes, il aura le résultat dans au maximum T secondes.

La dernière caractéristique de l'artifact est sa fonction. Cela est spécialement utile dans un système ouvert où il est nécessaire pour un agent de découvrir les artifacts susceptibles de l'aider à résoudre son problème. Ce n'est pas le type d'utilisation dont nous avons eu besoin dans l'application de localisation de bateaux et nous n'avons donc pas eu à spécifier la fonction de nos artifacts.

L'approche agent/artifact que nous avons utilisée dans cette application a permis de réduire considérablement la difficulté de la programmation des agents et donc d'autoriser une délibération beaucoup plus poussée que lorsque le calcul était partie intégrante de l'agent. La prise en compte des contraintes temporelles s'en est trouvée également très simplifiée.

5.4 Expérimentations

5.4.1 Avec et sans APRC

Nous avons lancé des expériences dans une version simplifiée d'Interloc pour exhiber les bénéfices liés à l'usage de l'APRC dans cette application. Nous avons comparé les résultats obtenus dans les cas suivants :

– *quand aucun mécanisme de coordination n'est utilisé pour gérer les ressources processeur allouées aux artifacts de calcul* Il n'y a aucune garantie que le calcul pour une mesure sera terminé avant qu'une nouvelle mesure arrive. Quand c'est le cas, l'artifact continue son calcul jusqu'à ce que le contrat de temps ait expiré. La nouvelle mesure est ignorée.

– *quand les services de l'APRC sont utilisés.* Quand l'APRC accepte d'ordonnancer une tâche, il garantit que celle-ci sera terminée avant l'arrivée de la prochaine mesure. Si la nouvelle tâche ne peut être insérée dans l'ordonnancement, la mesure correspondante est ignorée.

Nous calculons dans ces deux cas le pourcentage de mesures ignorées. Nous avons utilisés les valeurs suivantes, qui sont représentatives de ce que l'on peut avoir en réalité, pour les paramètres de l'application précédemment présentés :

– nombre de bateaux : 10,
– p_i : 5 à 30 secondes (en temps réel),
– PD : 2 secondes (en temps processeur)

Notre machine de référence utilise un processeur Intel Pentium 4 Mobile cadencé à 1,6 GHz et disposant de 768 Mo de mémoire.

Dans le premier cas, 42% des mesures sont ignorées. A cause de la désynchronisation des radars, il y a des moments où beaucoup de mesures arrivent à peu près au même moment, ce qui augmente le nombre de mesures ignorées. A d'autres moments, tous les calculs sur les mesures antérieures peuvent être terminés et aucune nouvelle mesure n'arrive. Le processeur peut ainsi être inutilisé pendant plusieurs secondes

L'utilisation des mécanismes de coordination de l'APRC apporte un gain significatif, puisqu'il n'y a plus que 7% des mesures qui sont ignorées. L'utilisation du processeur est optimisée car il est utilisé constamment à sa charge maximum.

5.4.2 Equilibrage de charge sur plusieurs processeurs

Nous avons réalisé d'autres expériences pour mettre en évidence l'échange d'information de charge processeur entre différents APRC déployés sur plusieurs machines et l'utilisation de ces informations dans la prise de décision de migration des agents.

Nous utilisons ici deux machines. La première est notre machine de référence utilisée dans la section précédente. La seconde machine dispose d'un processeur AMD K6-2 cadencé à 450 MHz et disposant de 256 Mo de mémoire. La préparation de l'expérimentation consiste à lancer un démon ALBA et un APRC sur chacune des machines. Les deux APRC se connaissent et peuvent s'échanger des informations par le réseau. La figure 5.5 montre une capture d'écran de l'interface qui nous permet de visualiser les informations dont ils disposent.

Nous reprenons les mêmes conditions que dans l'expérience précédente :
– l'APRC est activé,
– au lancement du système, tous les agents et artifacts démarrent sur la première machine,
– le nombre de bateaux est fixé à 10,
– les p_i vont de 5 à 30 secondes en temps réel,
– PD est fixé à 2 secondes en temps processeur.

La seule différence est que nous autorisons maintenant les agents à migrer de la première machine à la seconde (mais pas de la seconde à la première). La politique de migration est la suivante : dès qu'un agent se voit refuser l'allocation de ressources processeur par l'APRC, il lui demande de migrer, sans se préoccuper de la charge de la seconde machine. L'APRC accepte les migrations jusqu'à ce que les charges des deux machines soit égales. Une fois l'autorisation de migrer accordée, l'agent attend que l'ensemble des tâches de calcul exécutées par ses artifacts se terminent. Une fois qu'il a migré avec ses artifacts vers la seconde machine, il peut demander à son nouvel APRC d'allouer des ressources processeur pour ses tâches de calcul.



FIG. 5.5 – Capture d'écran de l'interface de visualisation en mode 2 machines.

Les résultats font apparaître que la migration d'un agent sur la seconde machine permet de limiter à moins de 1% le nombre de mesures ignorées et que la migration de deux agents permet d'obtenir 100% de mesures traitées.

Nous avons dans un second temps augmenté le nombre de bateaux jusqu'à 20. Dans cette situation, il y a un plus grand nombre d'agents qui migrent vers la seconde machine. Cependant, avec la politique de migration que nous avons choisie, ce sont les agents qui ont le moins de marge de manœuvre qui vont migrer vers la seconde machine (la marge de manœuvre est la différence entre le délai donné à la tâche et le temps nécessaire à son exécution). On va donc se retrouver dans une situation de déséquilibre : les bateaux pour lesquels les p_i sont grands vont être traités sur la première machine tandis que les bateaux pour lesquels les p_i sont petits vont être traités sur la seconde machine. Les calculs effectués ont moins de marge de manœuvre pour être ordonnancées et l'APRC est obligé de refuser énormément de nouvelles tâches de calcul. Cela est aggravé par la différence de rapidité entre les deux machines. Au final, on n'obtient que 22% de mesures traitées.

Nous avons enfin testé une politique de migration qui arrive à mieux sélectionner les agents qui doivent migrer pour limiter le nombre de mesures ignorées. Les agents ont toujours comme objectif de migrer dès qu'ils voient une de leur tâche refusée par l'APRC, mais ils demandent auparavant à l'APRC s'ils ont intérêt à le faire ou non. L'APRC concerné va demander à son homologue sur l'autre

machine la moyenne des marges de manœuvre pour les tâches qu'il a ordonnancées. Il la compare à la sienne et autorise l'agent à migrer si la migration permet de réduire l'écart entre les deux valeurs. Pour un même nombre d'agents ayant migré sur la seconde machine, nous obtenons maintenant un taux de 56% de mesures traitées.

Ces expérimentations montrent que les possibilités de migration offertes par la bibliothèque ALBA peuvent être mises à profit pour réaliser de l'équilibrage de charge dans nos SMA. Ceci n'était pas notre préoccupation principale dans cette étude mais les résultats encourageants ouvrent de nouvelles pistes de recherche dans ce sens, en particulier pour déterminer s'il est possible de trouver des politiques d'équilibrage de charge décentralisées qui respectent l'autonomie de décision des agents. Pour poursuivre dans la voie de l'équilibrage de charge par migration d'agents, on pourra s'inspirer des travaux sur la mobilité réalisés au LIP6 par Amal El Fallah Seghrouchni, Alexandru Suna et Gilles Klein [Klein *et al.*, 2004, Suna *et al.*, 2004, Klein, 2005, Suna, 2005, El Fallah Seghrouchni et Suna, 2005]. Cette équipe propose un langage de programmation d'agents nommé CLAIM (Computational Language for Autonomous, Intelligent and Mobile agents) et une plateforme d'exécution distribuée nommée SyMPA (System Multi-Platform of Agents). On retrouve dans ces travaux des motivations identiques à celles qui nous animent : proposer des techniques de programmation spécifiques aux systèmes multi-agents qui permettent de retrouver dans l"implémentation des agents les concepts utilisés au niveau de la modélisation des applications. Ces travaux se concentrent sur la mobilité d'agents intelligents. Les agents CLAIM peuvent exécuter plusieurs processus de manière concurrente et sont organisés en hiérarchies. Les primitives de mobilité, inspirées du calcul des ambients, permettent de quitter ou d'entrer dans une hiérarchie. CLAIM se place à un niveau d'abstraction auquel on peut définir une sémantique opérationnelle dans le but de pouvoir faire de la vérification des programmes. La plateforme SyMPA permet de déployer des hiérarchies d'agents sur un système distribué. Plusieurs hiérarchies peuvent être placées sur une même machine et l'on différencie donc la migration locale et la migration à distance.

5.5 Synthèse

Nous avons présenté la plateforme multi-agent développée au sein de Thales Division Aéronautique. Elle se base sur la bibliothèque ALBA embarquée dans chacun des agents. Elle est ainsi totalement décentralisée. Elle se focalise sur la conception d'agents programmés en Prolog et autorise le prototypage rapide de SMA en étant indépendant du modèle d'agent utilisé. Elle fournit les mécanismes nécessaires pour que les agents puissent migrer d'une machine à une autre sur un réseau.

Nous avons montré comment les artifacts ont pu être très facilement intégrés à nos systèmes multi-agents développés sur cette plateforme.

Nous avons présenté l'application Interloc de localisation passive de bateaux. Cette application est une bonne candidate pour mettre en œuvre les propositions faites dans cette thèse car les agents doivent respecter des délais imposés par la cadence à laquelle les capteurs envoient les informations à traiter. De plus, les traitements peuvent être longs et sont réalisés par un algorithme à contrat.

Les expérimentations que j'ai menées sur cette application réelle montrent que l'utilisation des artifacts de calcul permet d'obtenir une modélisation plus satisfaisante de l'application. Elles montrent également que la gestion par des APRC des ressources processeur disponibles permet d'apporter un gain substanciel par rapport au respect des délais imposés aux différentes tâches de calcul entreprises par les agents.

Chapitre 6

Conclusion

Une conclusion, c'est quand vous en avez assez de penser.
Herbert Fisher.

Nous avons posé dans cette thèse le problème du partage des ressources processeur dans les systèmes multi-agents qui ont des délibérations longues. Nous avons regardé ce problème à l'intersection de deux domaines : l'Intelligence Artificielle, car nos agents réalisent pour la plus grande part des tâches de calcul issus de ce domaine ; et le génie logiciel, puisque nous considérons que le paradigme agent est une évolution majeure pour une programmation simplifiée des systèmes complexes.

Dans ce cadre, nous avons identifié des propriétés importantes pour nos agents : l'autonomie, l'extraversion et la conscience du temps. Nous considérons que si nous perdons ces propriétés, nous perdons en même temps une grande partie des bénéfices apportés par le paradigme agent.

Les outils proposés par les systèmes d'exploitation pour gérer le partage des ressources processeur se placent à un niveau d'abstraction bien trop bas pour pouvoir être compatibles avec les exigences citées précédemment au niveau agent. Par exemple, si l'ordonnancement des tâches est uniquement décidé par le système d'exploitation, les agents lui sont totalement soumis et ne peuvent exercer aucune autonomie sur leur propre utilisation des ressources processeur. Il convient donc de proposer une interface entre ces deux niveaux d'abstraction.

Je soutiens la thèse qu'il est nécessaire pour cela d'introduire un nouveau type d'entité dans les systèmes multi-agents pour prendre en compte au mieux les spécificités des SMA que nous voulons construire. Le concept d'artifact introduit dans [Omicini *et al.*, 2004] permet justement d'intégrer dans les SMA des entités utilisées par les agents comme des outils les aidant à atteindre leurs buts.

J'introduis une spécialisation de ce concept pour les délibérations longues. La nouvelle classe d'artifacts ainsi définie, les artifacts de calcul, permet aux agents de conserver leurs propriétés d'extraversion et de conscience du temps en déléguant leurs délibérations longues à des outils de calculs

dédiés. Ceux-ci disposent de modes d'emploi qui permettent aux agents de les instancier et de contrôler l'exécution des algorithmes qu'ils encapsulent. Les modes d'emploi décrivent également les propriétés temporelles des algorithmes utilisés, ce qui peut permettre aux agents de raisonner dessus. J'ai également introduit un mécanisme de conversion d'un mode d'emploi d'artifact de calcul vers une structure de tâche TÆMS, qui propose d'autres types de raisonnements.

Le formalisme à base d'algèbre de processus proposé dans [Omicini *et al.*, 2004] est extensible et nous avons proposé des extensions suffisantes pour gérer le temps dans cette étude. Nous proposons également une extension qui autorise l'utilisation d'informations sur l'historique des actions et perceptions liées au mode d'emploi pour activer ou désactiver des branches de celui-ci. Nous avons cependant remarqué que les possibilités d'extension sont limitées et qu'il sera difficile de plus flexibiliser ce langage et de permettre en même temps des raisonnements plus fins. De plus, un mode d'emploi décrit dans ce langage n'est pas directement compréhensible par l'agent et l'on doit fournir un lien sémantique entre les données du mode d'emploi et la base de connaissances de l'agent. Ceci n'est pas satisfaisant. Il serait plus intéressant que le formalisme utilisé permette de décrire des modes d'emploi auto-descriptifs. Nous avons également identifié d'autres points qui manquent encore pour gérer les interactions du couple agent/artifact de manière satisfaisante. En particulier, la découverte de nouveaux artifacts de calculs à l'exécution et leur mise en œuvre de manière automatique grâce à la compréhension de leur mode d'emploi est un problème difficile pour lequel le formalisme actuellement utilisé ne propose pas de solution.

J'ai proposé dans un second temps des outils basés sur les artifacts pour gérer le partage des ressources processeur en prenant en compte toutes les contraintes que pose pour cela le paradigme agent. En effet, il est important que les agents puissent conserver leur autonomie de décision, même s'il existe une entité (l'Artifact de Partage des Ressources de Calcul, APRC) qui calcule l'ordonancement des tâches sur le processeur. Je propose un algorithme d'ordonnancement qui permet aux artifacts de calcul de respecter des contraintes de calcul, tout en permettant de considérer que les agents sont actifs en permanence.

Le système, initialement développé pour un unique processeur, peut être étendu à plusieurs machines en échangeant au niveau des différents APRC les informations de charge de leur processeur et en autorisant les agents à migrer d'une machine à l'autre. Cela ouvre des perspectives quant à l'utilisation de nos techniques dans le cadre de l'équilibrage de charges sur un réseau de machines.

Les propositions précédentes ont été évaluées sur la plate-forme multi-agent ALBA développée au sein de Thales Division Aéroportée. Nous avons implanté dans cette plate-forme un modèle d'agent basé sur des Fils De Raisonnement, qui permettent de gérer de manière déterministe un ensemble de contextes de communication et de raisonnements.

J'ai réimplémenté l'application Interloc de localisation passive de bateaux en utilisant les artifacts de calcul et l'APRC. La nouvelle version apporte une meilleure gestion des temps de calcul, ainsi que des gains susbtanciels au niveau de la clarté de la modélisation.

Les propositions faites dans cette thèse pourront à l'avenir être utilisées dans les systèmes multi-agents pour lesquels les durées des calculs entrepris par les agents posent problème et/ou lorsque

l’on veut pouvoir imposer des délais aux tâches de calcul. Elles permettent de réduire le fossé qui existe entre le niveau agent et le niveau système, tout en ayant la possibilité de considérer les agents selon un haut niveau d’abstraction. Les artifacts de calcul sont un nouveau type d’entités que l’on peut intégrer dans les systèmes multi-agents existants. Il faut pour cela décrire le fonctionnement des algorithmes qu’ils vont encapsuler dans le formalisme du langage de mode d’emploi et gérer au niveau des agents la délégation des tâches de calcul en suivant le mode d’emploi précédemment décrit. L’APRC est quant à lui un artifact de coordination réutilisable permettant de gérer les ressources processeur affectées aux agents et aux artifacts de manière générique.

Chapitre 7

Perspectives

On ne fait jamais attention à ce qui a été fait ; on ne voit que ce qui reste à faire.
Marie Curie.

Lorsqu'il n'y a plus rien à faire, que faites-vous ?
Koan zen.

Nous avons utilisé pour cette étude une approche ascendante. Cette thèse nous a permis de mettre en place les couches les plus basses permettant de gérer au mieux les délibérations longues dans nos agents. Nous pouvons déjà identifier quelques améliorations qui seront nécessaires dans ce que nous venons de proposer avant de pouvoir construire les couches supérieures. Cela concerne le mode d'emploi des artifacts et le modèle d'agent que nous utilisons. Nous pourrons ensuite passer à une étude beaucoup plus globale sur la gestion du temps dans les systèmes multi-agents qui utilisent des artifacts de calcul. Nous pensons également que notre approche utilisant les artifacts permet de nous rapprocher d'autres communautés travaillant sur d'autres types de systèmes distribués.

7.1 Mode d'emploi des artifacts

Nous avons déjà soulevé dans la section 3.4.3.5 un certain nombre de limitations du langage actuellement utilisé pour décrire les modes d'emploi des artifacts. Nous aimerions être capables de décrire des conseils d'utilisation en complément des règles d'utilisation qu'il est obligatoire de respecter. Nous aimerions également élargir la sémantique du langage pour utiliser certaines possibilités du langage TÆMS.

Nos agents utilisent pour l'instant les artifacts de manière inconsciente. Nous aimerions qu'ils soient capables d'identifier le besoin d'utiliser un outil, de les découvrir dynamiquement et d'interpréter de nouveaux modes d'emploi d'artifacts.

7.2 Modèle d'agent

Le stage de Master Recherche de Benjamin Devèze a permis d'identifier des fonctionnalités manquantes ou à améliorer dans le modèle d'agent que nous utilisons actuellement [Devèze, 2006]. Celles-ci concernent :

- *la gestion des signaux.* Les seuls signaux qui parviennent actuellement aux agents sont les messages qu'ils reçoivent. Il serait intéressant de pouvoir gérer d'autres types de signaux hautement prioritaires à la façon des interruptions systèmes ou encore donner la possibilité à l'agent de se programmer l'envoi d'un signal d'attention à une date donnée.
- *la gestion de l'agenda.* Cela concerne la possibilité de pouvoir facilement programmer des actions pour le futur et qu'elles se déclenchent au bon moment de manière automatique, comme cela est fait dans Agent-0.
- *des moyens pour se constituer des bibliothèques de fils de raisonnement réutilisables.*
- *la gestion de la mémoire liée à chacun des fils de raisonnement et à l'agent complet.*

7.3 Gestion du temps dans les systèmes multi-agents

Une fois que notre modèle d'agent disposera des fonctionnalités décrites ci-dessus, nous pourrons mettre en place le niveau méta permettant de raisonner sur les actions de l'agent. Les capacités intelligentes des agents à développer concerneront alors :

- *les buts de l'agent.* Ceux-ci doivent apparaître de manière explicite, contrairement à ce qui est fait actuellement (les buts sont exprimés de manière implicite dans le code). L'agent pourrait ainsi manipuler ses buts et raisonner dessus.
- *l'utilisation et le raisonnement sur les plans de l'agent.* Les agents devraient être capables de manipuler des plans hypothétiques (les instancier, les évaluer, les comparer, les mettre en œuvre). Ils devraient également être capables de raisonner sur des plans qui leur sont fournis.
- *l'utilisation des informations sur l'historique des actions et des perceptions.* Ces informations, souvent utilisées pour l'apprentissage, nous serviraient plutôt comme aide aux décisions de l'agent.

7.4 Rapprochement avec d'autres communautés

Nous devrions assister dans les années qui viennent à un rapprochement, déjà amorcé, entre différentes communautés. C'est par exemple le cas entre la communauté agent et la communauté composant. Un agent peut être constitué de plusieurs composants ou un composant peut disposer de capacités cognitives. Nous avons le sentiment que les réponses que nous apportons pour les systèmes d'agents avec nos travaux sur les artifacts se placent au bon niveau d'abstraction pour pouvoir être réutilisées et pour ajouter de l'intelligence aux systèmes à composants ou aux Web Services par exemple. Un

rapprochement serait également possible avec la communauté qui travaille sur les architectures dirigées par les modèles (MDA/MDE) et qui œuvre sur des spécifications déclaratives des systèmes. Cette communauté utilise les modèles pour spécifier séparément l'application et l'architecture matérielle qui va la supporter. Le code final est obtenu de manière automatique en mappant petit à petit un modèle software sur un modèle hardware. Le concept d'artifact devrait également pouvoir être adapté à cette approche pour obtenir des fonctionnalités symétriques côté software permettant des raisonnements sur les modèles.

Le domaine des calculs intensifs sur grilles de calcul est également un domaine dont les techniques de base sont matures (déploiement et équilibrage des calculs, hétérogénéité du matériel, . . .). Les chercheurs dans ce domaine sont maintenant demandeurs de techniques permettant d'ajouter une couche d'intelligence à leurs systèmes et de décentraliser le contrôle des calculs. Là encore, les travaux sur les artifacts de calculs pourraient être transférés à ce type d'applications.

Bibliographie

[Alur *et al.*, 1998] ALUR, R., HENZINGER, T. A. et KUPFERMAN, O. (1998). Alternating-time temporal logic. *Lecture Notes in Computer Science*, 1536:23–60.

[Arisha *et al.*, 1999] ARISHA, K. A., OZCAN, F., ROSS, R., SUBRAHMANIAN, V. S., EITER, T. et KRAUS, S. (1999). IMPACT : A platform for collaborating agents. *IEEE Intelligent Systems*, 14(2):64–72.

[Barbuceanu et Fox, 1995] BARBUCEANU, M. et FOX, M. S. (1995). Cool : A language for describing coordination in multiagent systems. *In* LESSER, V. et GASSER, L., éditeurs : *Proceedings of the First International Conference oil Multi-Agent Systems (ICMAS-95)*, pages 17–24, San Francisco, CA, USA. AAAI Press.

[Basseur, 2005] BASSEUR, M. (2005). *Conception d'algorithmes coopératifs pour l'optimisation multi-objectif : Application aux problèmes d'ordonnancement de type Flow-shop*. Thèse de doctorat, Université Lille 1.

[Boddy et Dean, 1989] BODDY, M. et DEAN, T. (1989). Solving time-dependent planning problems. Rapport technique.

[Botella et Taillibert, 1993] BOTELLA, B. et TAILLIBERT, P. (1993). Interlog : Constraint logic programming on numeric intervals. *3rd Int. Workshop on Software Engineering, Artificial Intelligence and Expert Systems for High Energy and Nuclear Physics.*

[Bussman et Demazeau, 1994] BUSSMAN, S. et DEMAZEAU, Y. (1994). An agent model combining reactive and cognitive capabilities. *In Intelligent Robots and Systems '94. Advanced Robotic Systems and the Real World*.

[Buytaert, 2005] BUYTAERT, K. (2005). openmosix past, present and future. *In UKUUG'05*.

[Cappello *et al.*, 2005] CAPPELLO, F., DJILALI, S., FEDAK, G., HERAULT, T., MAGNIETTE, F., NORI, V. et LODYGENSKY, O. (2005). Computing on large-scale distributed systems : Xtrem web architecture, programming models, security, tests and convergence with grid. *Future Gener. Comput. Syst.*, 21(3):417–437.

[Carabelea *et al.*, 2003] CARABELEA, C., BOISSIER, O. et FLOREA, A. (2003). Autonomy in multi-agent systems : A classification attempt. *In* NICKLES, M., ROVATSOS, M. et WEISS, G., éditeurs : *Agents and Computational Autonomy*, volume 2969 de *Lecture Notes in Computer Science*, pages 103–113. Springer.

[Cheeseman *et al.*, 1991] CHEESEMAN, P., KANEFSKY, B. et TAYLOR, W. M. (1991). Where the Really Hard Problems Are. *In Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, IJCAI-91, Sidney, Australia*, pages 331–337.

[Culler *et al.*, 1999] CULLER, D. E., SINGH, J. P. et GUPTA, A. (1999). *Parallel Computer Architecture - A hardware/software approach*. Morgan Kaufmann Publishers, Inc.

[Dean et Boddy, 1988] DEAN, T. et BODDY, M. S. (1988). An analysis of time-dependent planning. *In AAAI*, pages 49–54.

[Decker, 1995] DECKER, K. (1995). Environment Centered Analysis and Design of Coordination Mechanisms. *Ph.D. Thesis, Department of Computer Science, University of Massachusetts, Amherst*.

[Delhay, 2000] DELHAY, A. (2000). *Contribution à l'algorithmique anytime : Contrôle et Conception*. Thèse de doctorat, Université Lille 1.

[Demazeau, 1995] DEMAZEAU, Y. (1995). From interactions to collective behaviour in agent-based systems. *In Proceedings of the First European conference on cognitive science*, pages 117–132, Saint Malo, France.

[Devigne *et al.*, 2004] DEVIGNE, D., MATHIEU, P. et ROUTIER, J.-C. (2004). Planning for spatially situated agents. *In IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'04)*, pages 385–388. IEEE Press.

[Devigne *et al.*, 2005a] DEVIGNE, D., MATHIEU, P. et ROUTIER, J.-C. (2005a). Interaction-based approach for games agents. *In Proceedings of the 19th Europeen Conference on Modelling and Simulation, Simulation in Wider Europe - ECMS 2005*, pages 705–714.

[Devigne *et al.*, 2005b] DEVIGNE, D., MATHIEU, P. et ROUTIER, J.-C. (2005b). Simulation de comportements centrée interaction. *In Journées Francophones sur les Systèmes Multi-Agents (JFSMA'05) , Calais, France*. Hermès-Lavoisier.

[Devigne *et al.*, 2005c] DEVIGNE, D., MATHIEU, P. et ROUTIER, J.-C. (2005c). Team of cognitive agents with leader : how to let them acquire autonomy. *In Proceedings of 2005 IEEE Symposium on Computational Intelligence and Games*.

[Devèze, 2006] DEVÈZE, B. (2006). Programmation d'agents intelligents - vers une refonte des fils de raisonnement. Mémoire de D.E.A., Université Pierre et Marie Curie.

[Devèze *et al.*, 2006] DEVÈZE, B., CHOPINAUD, C. et TAILLIBERT, P. (2006). Alba : a generic library for programming mobile agents with prolog. *In AAMAS'06*.

[Dinont, 2003] DINONT, C. (2003). Un langage et un modèle d'agent. Mémoire de D.E.A., Laboratoire d'Informatique Fondamentale de Lille - Université Lille 1.

[Dinont *et al.*, 2006a] DINONT, C., DRUON, E., MATHIEU, P. et TAILLIBERT, P. (2006a). Artifacts for time-aware agents. *In AAMAS'06*.

[Dinont *et al.*, 2006b] DINONT, C., DRUON, E., MATHIEU, P. et TAILLIBERT, P. (2006b). CPU sharing for autonomous time-aware agents. *In COGIS'06*.

[Dix *et al.*, 2001] DIX, J., KRAUS, S. et SUBRAHMANIAN, V. S. (2001). Temporal agent programs. *Artificial Intelligence*, 127(1):87–135.

[Eiter et Subrahmanian, 1999] EITER, T. et SUBRAHMANIAN, V. S. (1999). Heterogeneous active agents, ii : Algorithms and complexity. *Artif. Intell.*, 108(1-2):257–307.

[El Fallah Seghrouchni et Suna, 2005] EL FALLAH SEGHROUCHNI, A. et SUNA, A. (2005). *Multi-Agent Programming : Languages, Platforms and Applications*, volume 15 de *Multiagent Systems, Artificial Societies, and Simulated Organizations*, chapitre 4 : CLAIM and SyMPA : a programming environment for intelligent and mobile agents. Springer.

[Faure, 1979] FAURE, R. (1979). *Précis de recherche opérationnelle*. Dunod.

[Ferber, 1995] FERBER, J. (1995). *Les Systèmes Multi-Agents, Vers une intelligence collective*. InterEditions.

[Ferguson, 1992] FERGUSON, I. A. (1992). Touring machines : Autonomous agents with attitudes. *Computer*, 25(5):51–55.

[Fisher *et al.*, 2005] FISHER, M., GABBAY, D. M. et VILA, L., éditeurs (2005). *Handbook of Temporal Reasoning in Artificial Intelligence*. Elsevier.

[Fokkink, 2000] FOKKINK, W. (2000). *Introduction to Process Algebra*. Springer-Verlag.

[Foster *et al.*, 2004] FOSTER, I., JENNINGS, N. R. et KESSELMAN, C. (2004). Brain meets brawn : Why grid and agents need each other. *In AAMAS '04 : Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 8–15, Washington, DC, USA. IEEE Computer Society.

[Foster *et al.*, 2002] FOSTER, I., KESSELMAN, C., NICK, J. et TUECKE, S. (2002). The physiology of the grid : An open grid services architecture for distributed systems integration.

[Gamma *et al.*, 1993] GAMMA, E., HELM, R., JOHNSON, R. et VLISSIDES, J. (1993). Design patterns : Abstraction and reuse of object-oriented design. *Lecture Notes in Computer Science*, 707:406–431.

[Garvey et Lesser, 1996] GARVEY, A. et LESSER, V. R. (1996). Design-to-time scheduling and anytime algorithms. *SIGART Bull.*, 7(2):16–19.

[Graham, 2001] GRAHAM, J. R. (2001). *Real-time Scheduling in Distributed Multiagent Systems*. Thèse de doctorat, University of Delaware.

[Graham *et al.*, 2003] GRAHAM, J. R., DECKER, K. S. et MERSIC, M. (2003). Decaf - a flexible multi agent system architecture. *Autonomous Agents and Multi-Agent Systems*, 7(1-2):7–27.

[Grass, 1996] GRASS, J. (1996). Reasoning about computational resource allocation. *Crossroads*, 3(1):16–20.

[Guessoum et Dojat, 1996] GUESSOUM, Z. et DOJAT, M. (1996). A real-time agent model in an asynchronous-object environment. *In* van HOE, R., éditeur : *Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*.

[Hewitt *et al.*, 1973] HEWITT, C., BISHOP, P. et STEIGER, R. (1973). A universal modular actor formalism for artificial intelligence. *In IJCAI*, pages 235–245.

[Horling, 1999] HORLING, B. (1999). http ://mas.cs.umass.edu/research/taems/white/.

[Horvitz et Rutledge, 1991] HORVITZ, E. et RUTLEDGE, G. (1991). Time-dependent utility and action under uncertainty. *In Proceedings of the seventh conference (1991) on Uncertainty in artificial intelligence*, pages 151–158, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Horvitz, 1990] HORVITZ, E. J. (1990). Reasoning about beliefs and actions under computational resource constraints. *In* HENRION, M., SHACHTER, R. D., KANAL, L. N. et LEMMER, J. F., éditeurs : *Uncertainty in Artificial Intelligence 5*, pages 301–324. Elsevier Science Publishers B.V., North Holland.

[Ingrand *et al.*, 1992] INGRAND, F. F., GEORGEFF, M. P. et RAO, A. S. (1992). An architecture for real-time reasoning and system control. *IEEE Expert : Intelligent Systems and Their Applications*, 07(6):34–44.

[Jennings *et al.*, 1998] JENNINGS, N. R., SYCARA, K. et WOOLDRIDGE, M. (1998). A roadmap of agent research and development. *Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):7–38.

[Jonquet *et al.*, 2006] JONQUET, C., DUGÉNIE, P. et CERRI, S. A. (2006). Intégration orientée service des modèles grid et multi-agents. *In JFSMA'06 : Journées Francophones sur les Systèmes Multi-Agents*. Hermes.

[Joseph et Kawamura, 2001] JOSEPH, S. et KAWAMURA, T. (2001). *Agent Engineering*, chapitre Why Autonomy Makes the Agent. World Scientific Publishing.

[Kaptelinin *et al.*, 1995] KAPTELININ, V., KUUTTI, K. et BANNON, L. J. (1995). Activity theory : Basic concepts and applications. *In EWHCI*, pages 189–201.

[Klein, 2005] KLEIN, G. (2005). *Distributed Multiagent Systems and Multiagent Distributed Systems*. Thèse de doctorat, Université Paris-Dauphine.

[Klein *et al.*, 2004] KLEIN, G., SUNA, A. et EL FALLAH SEGHROUCHNI, A. (2004). Resource sharing and load balancing based on agent mobility. *In ICEIS (4)*, pages 350–355.

[Knuth, 1997] KNUTH, D. E. (1997). *The Art Of Computer Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Kuwabara *et al.*, 1995] KUWABARA, K., OSATO, N. et ISHIDA, T. (1995). Agentalk : Describing multiagent coordination protocols with inheritance. *In TAI '95 : Proceedings of the Seventh International Conference on Tools with Artificial Intelligence*, page 460, Washington, DC, USA. IEEE Computer Society.

[Lacouture et Aniorté, 2006] LACOUTURE, J. et ANIORTÉ, P. (2006). Vers l'adaptation dynamique de services : des composants monitorés par des agents. *In 2eme journée multi-agent et composant*.

[Lee, 2006] LEE, E. A. (2006). The problem with threads. *Computer*, 39(5):33–42.

[Lhomme, 1993] LHOMME, O. (1993). Consistency techniques for numeric CSPs. *In IJCAI'93*.

[Lhomme *et al.*, 1994] LHOMME, O., GOTLIEB, A. et RUEHER, M. (1994). Dynamic optimization of interval narrowing algorithms. *Journal of Logic Programming*, 19-20.

[Liu et Layland, 1973] LIU, C. L. et LAYLAND, J. W. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61.

[Luck et d'Inverno, 2001] LUCK, M. et D'INVERNO, M. (2001). Autonomy : A nice idea in theory. *Lecture Notes in Computer Science*, 1986.

[Marson, 2006] MARSON, P.-E. (2006). Artifacts - abstractions pour la modélisation et la mise en œuvre d'un environnement au sein d'un système multi-agents. Mémoire de D.E.A., Université Paris Dauphine.

[Mathieu et Picault, 2005] MATHIEU, P. et PICAULT, S. (2005). Towards an interaction-based design of behaviors. *In European workshop on Multi-Agent systems (EUMAS'05)*.

[Mathieu et Picault, 2006] MATHIEU, P. et PICAULT, S. (2006). Vers une représentation des comportements centrée interactions. *In RFIA'06*.

[McCabe et Clark, 1995] MCCABE, F. G. et CLARK, K. L. (1995). April – agent process interaction language. *In* WOOLDRIDGE, M. et JENNINGS, N. R., éditeurs : *Intelligent Agents : Theories, Architectures, and Languages (LNAI volume 890)*, pages 324–340. Springer-Verlag : Heidelberg, Germany.

[Milner, 1982] MILNER, R. (1982). *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Moszkowski, 1986] MOSZKOWSKI, B. (1986). *Executing temporal logic programs*. Cambridge University Press, New York, NY, USA.

[Muller et Pischel, 1993] MULLER, J. et PISCHEL, M. (1993). The agent architecture InteRRaP : Concept and application.

[Odell, 2002] ODELL, J. (2002). Objects and agents compared. *Journal of Object Technology*, 1(1): 41–53.

[Omicini *et al.*, 2005] OMICINI, A., RICCI, A. et VIROLI, M. (2005). Agens faber : Toward a theory of artifacts for MAS. *In Electronic Notes in Theoretical Computer Sciences. First International Workshop "Coordination and Organization" (CoOrg05), COORDINATION 2005*.

[Omicini *et al.*, 2004] OMICINI, A., RICCI, A., VIROLI, M., CASTELFRANCHI, C. et TUMMOLINI, L. (2004). Coordination artifacts : Environment-based coordination for intelligent agents. *In AAMAS'04*.

[OpenMP,] OPENMP. *http ://openmp.org*.

[Ricci *et al.*, 2005] RICCI, A., VIROLI, M. et OMICINI, A. (2005). Programming MAS with artifacts. *In ProMAS'05*.

[Ricci *et al.*, 2006] RICCI, A., VIROLI, M. et OMICINI, A. (2006). CArtAgO : An infrastructure for engineering computational environments in MAS. *In* WEYNS, D., PARUNAK, H. V. D. et MICHEL, F., éditeurs : *3rd International Workshop "Environments for Multi-Agent Systems" (E4MAS 2006)*, pages 102–119, AAMAS 2006, Hakodate, Japan.

[Shoham, 1993] SHOHAM, Y. (1993). Agent-oriented programming. *Artificial Intelligence*, 60(1): 51–92.

[Subrahmanian *et al.*, 2000] SUBRAHMANIAN, V. S., BONATTI, P., DIX, J., EITER, T., KRAUS, S., OZCAN, F. et ROSS, R. (2000). *Heterogeneous Agent Systems*. MIT Press/AAAI Press, Cambridge, MA, USA.

[Suna, 2005] SUNA, A. (2005). *CLAIM et SyMPA : Un environnement pour la programmation d'agents intelligents et mobiles*. Thèse de doctorat, Université Paris 6.

[Suna *et al.*, 2004] SUNA, A., KLEIN, G. et EL FALLAH SEGHROUCHNI, A. (2004). Using mobile agents for resource sharing. *In IAT*, pages 389–392. IEEE Computer Society.

[Vincent *et al.*, 2001] VINCENT, R., HORLING, B., LESSER, V. R. et WAGNER, T. (2001). Implementing Soft Real-Time Agent Control. *Proceedings of the 5th ICAA*, pages 355–362.

[Viroli et Ricci, 2004] VIROLI, M. et RICCI, A. (2004). Instruction-based semantics of agent mediated interaction. *In AAMAS'04*.

[Wagner, 2000] WAGNER, T. (2000). *Toward Quantified, Organizationally Centered, Decision Making and Coordination*. Thèse de doctorat, University of Massachusetts.

[Wagner et Lesser, 2000] WAGNER, T. et LESSER, V. R. (2000). Design-to-Criteria Scheduling : Real-Time Agent Control. *Proceedings of AAAI 2000 Spring Symposium on Real-Time Autonomous Systems*.

[Williamson *et al.*, 1996] WILLIAMSON, M., DECKER, K. et SYCARA, K. (1996). Unified information and control flow in hierarchical task networks. *In Theories of Action, Planning, and Robot Control : Bridging the Gap : Proceedings of the 1996 AAAI Workshop*, pages 142–150, Menlo Park, California. AAAI Press.

[Wooldridge *et al.*, 2000] WOOLDRIDGE, M., JENNINGS, N. R. et KINNY, D. (2000). The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312.

[Zilberstein et Mouaddib, 1999] ZILBERSTEIN, S. et MOUADDIB, A. I. (1999). Reactive control of dynamic progressive processing. *In Proceedings of the 16th IJCAI*, pages 1268–1273. Morgan Kaufmann Publishers Inc.